NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DISTRIBUTED RELATIONAL DATABASE SYSTEM OF OCCASIONALLY CONNECTED DATABASES

by

Pavel Bielecki

March 2000

Thesis Advisor: C. Thomas Wu
Second Reader: Chris Eagle

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Distributed Relational Database System of Occasionally Connected Databases	
6. AUTHOR(S) Bielecki, Pavel	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	

13. ABSTRACT *(maximum 200 words)*

The Troop Command at the Presidio of Monterey requires an information system that will provide timely and accurate data about all serviced troop activities with students and permanent party stationed at the Defense Language Institute Foreign Language Center. Data sources that could provide required information already exist, but are physically spread over the Presidio, are maintained in diverse formats, and are not interconnected. Some data sources, maintained by other activities located at the Presidio, are available on the Campus Area Network. As new technologies emerged, it became possible to integrate all available data sources into a heterogeneous distributed information system, in which some information will be shared, while other information will be under some degree of local control. This thesis studies the feasibility of such an information system, and proposes one possible implementation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Distributed Database, Heterogeneous Database System, PowerBuilder 7, SQL Server 7	192
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

DISTRIBUTED RELATIONAL DATABASE SYSTEM OF OCCASIONALLY CONNECTED DATABASES

Pavel Bielecki
Computer Specialist
Directorate of Information Management
Defense Language Institute Foreign Learning Center
and Presidio of Monterey

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: _____
Pavel Bielecki

Approved by: _____
C. Thomas Wu, Thesis Advisor

_____
Chris Eagle, Second Reviewer

_____
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

The Troop Command at the Presidio of Monterey requires an information system that will provide timely and accurate data about all serviced troop activities with students and permanent party stationed at the Defense Language Institute Foreign Language Center. Data sources that could provide required information already exist, but are physically spread over the Presidio, are maintained in diverse formats, and are not interconnected. Some data sources, maintained by other activities located at the Presidio, are available on the Campus Area Network. As new technologies emerged, it became possible to integrate all available data sources into a heterogeneous distributed information system, in which some information will be shared, while other information will be under some degree of local control. This thesis studies the feasibility of such an information system, and proposes one possible implementation.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my thesis advisors, Professor Thomas C. Wu and LCDR Chris Eagle, for their guidance and insightful comments.

I would also like to express my gratitude to the Command Group of the Presidio of Monterey and the management of the Directorate of Information Management for giving me the opportunity to pursue a degree in computer science.

I want to express special thanks to my mentor, Pat Golden, for sparking my interest in the field of knowledge systems, and for the generosity with which he continues to share his expertise.

I dedicate this thesis to my children, Vit, Adela, and Julie. I hope it will inspire them to pursue higher education.

I. INTRODUCTION

A. BACKGROUND

Numerous local databases are maintained simultaneously by military units at the Presidio of Monterey (POM). Each database implements its own proprietary database schema, and data are stored in a variety of formats (Paradox, Excel, MS Access, proprietary file system, Word document, etc.). Often, one local database consists of several unconnected mini databases. Because of such diversity, integration of these databases into one interconnected system is virtually impossible. As a result, data cannot be shared between units, nor can be queried by global users at the troop command level. In order to allow querying the data by global users, and to allow sharing of data between the units, these databases need to be integrated into one database system.

A standard database, named Military Student Database (MILDB), that contains all logical data items needed to be maintained by each unit, has been designed and locally implemented at some units. MILDB users continuously update and query their local databases. However, many data items, normally entered into MILDB at the unit level, already exist in a networked database maintained elsewhere at the Presidio of Monterey. In order to allow effective data sharing between these local databases and networked databases, they need to be integrated into one heterogeneous, distributed database system. Selection of appropriate database management systems that will support heterogeneous queries will be proposed. Since only some MILDB users have permanent access to Campus Area Network (CAN), while other users will connect to CAN only occasionally via modem, MILDB databases need to synchronized. Mechanism

for reliable, bi-directional synchronization of heterogeneous databases will be, therefore, proposed.

B. SCOPE

The scope of this thesis is to design and implement an information system, consisting of a multitude of local MILDB databases occasionally connected to a central MILDB database. This database system will be supported by a single interface which will be installed on every user's workstation, and will have the capability to communicate with both the central and local MILDB databases.

It will be taken into account that different system and performance requirements will apply for the central and local databases. The first step is the selection of appropriate database management system for each database. Chapter III, part B addresses this issue. The central database will contain data from all Units and will also contain other information, not related to MILDB. The second step, described in section E and F of Chapter III of this thesis, is to design and implement a strict system of access control which will provide information to users on need-to-see basis. The third step is to develop a user friendly and intuitive interface that will provide means for easy data updates, quick retrieval of data into canned or custom reports, and for bi-directional data synchronization. Section V of this thesis documents the development of the MILDB application interface.

C. ORGANIZATION OF THESIS

Chapter I: Introduction. Describes the project and identifies major areas of concern. Indicates steps that will be taken to resolve the project.

Chapter II: Information System Analysis. Provides analysis of hardware and software requirements on a workstation of a typical MILDB user. Detailed requirements for MILDB graphical user interface are also specified.

Chapter III: Knowledge Systems. Provides a general overview of database management systems (DBMSs), and major components of relational DBMS model. Selection of DBMS for local MILDB and the central database is proposed and justified. Data security is studied, and mechanism for controlling data access is proposed.

Chapter IV: Client Server Architecture. Applicability of two-tier and multi-tier architecture to MILDB is studied. Selection of client server model is made and justified.

Chapter V: Developing Database Application in PowerBuilder 7. Provides an overview of major features in PowerBuilder 7 for Windows. Describes how was PowerBuilder applied to develop the MILDB application interface, and to perform database transactions. Database connectivity through major database interfaces is also studied.

Chapter VI: Conclusion. Summarizes lessons learned, and proposes areas for further research.

THIS PAGE INTENTIONALLY LEFT BLANK

II. INFORMATION SYSTEM ANALYSIS

A. HARDWARE

Initially, local record keeping at Unit level was performed on PCs fitted with Intel 386 or 486 processor, 4MB to 8MB RAM, and 80 to 120 MB hard drive. 14in monitor was a standard. This was sufficient to run DOS, or simple Windows-based applications running in Windows 3.1 environment. At the time of implementation of the thesis, all potential users of MILDB were equipped with workstations having the following parameters:

Processor:	Pentium 266 MHz
RAM:	64 MB
Hard Drive:	8 GB
(Modem):	33.3 K
(Network card):	Ethernet 3COM 10-100
Monitor:	17"

Workstations are connected to Campus Area Network (CAN) via network card, or via modem. The central database was at the time of implementation of this thesis running on HP server fitted with four processors.

In order to take advantage of new hardware capabilities of the typical workstation, the original version of MILDB was redesigned and upgraded to PowerBuilder 7. This, in turn, set new minimum hardware requirements for workstations intended to run MILDB:

Processor:	Pentium 90 MHz +
RAM:	32 MB
Hard Drive:	7 MB
Monitor:	17"
Operating System:	WIN 95+ or WINT NT 4+

B. SOFTWARE

The typical MILDB user uses PC with the following software preinstalled:

Operating System:	Windows NT 4.0 Workstation
	Service Pack 4 or higher
Other Essential Software:	Office 97 with
	Microsoft Access 97
Hard Drive	32 bit ODBC Administrator

It is apparent that the typical user of MILDB at POM works on a workstation that exceeds the minimum hardware and software requirements for running a PowerBuilder 7 application.

C. INTERFACE

1. Goal

Develop a Windows based, event driven GUI as the sole means of communication with a database to be used by military personnel with computer skills ranging from novice to expert, who should become at least 80% proficient in using all GUI's features (i.e., be able to readily locate specific function feature and use it effectively) after less than 30 min of introductory briefing.

2. Features

- Maintains records of service members in three distinct areas: Administration, Physical Training, and Dormitory Assignment. Maintaining records includes: create new or locate existing record of service member, enter/update data, deactivate or permanently delete a record.

- Generates and displays canned (pre-designed) or ad hoc query reports.

- Prints a report, or exports report as a text file.

- Allows display/modification of data of personnel grouped by individual units (viewed only by personnel from the unit), and also make all data from all units available to global users at troop command level. At troop command level, also generate summary reports across all units.

- Minimizes the need of typing by providing optional selection from list of entries whenever practical/functional.

3. User Analysis

Users of the interface will be military personnel with computer skills ranging from novice to advanced level., ranking from enlisted personnel to senior officers. Users with lower ranks, who will use MILDB daily, may fluctuate relatively often (every few months). Officers with higher ranks may be using the system only sporadically. Therefore, the interface has to be simple and intuitive enough, so that the primary users (enlisted personnel) can use the system effectively after less than 30 min of training, and occasional users (officers) will be able to reach the desired information easily without any outside help.

Enlisted personnel (local users) will perform data entry and maintenance, and will generate reports at the unit level. Senior personnel (global users) will mostly generate summary reports.

No specific typing skills are required. It is assumed that all users will, as a minimum, have high school diploma.

4. Task Analysis

a. Display interface, retrieve data in following record categories:

(1) Admin/Biographical

- Inprocessing (biographical data, previous training, etc.).
- Individual Information (pregnancy counseling, family care, chapter discharge).
- Individual History (training, flags, qualifications, disciplinary actions).

- Outprocessing (deactivate record, permanently delete record).
- Reports/Schedules (rosters, reports, training plans).

(2) Physical Training

- Weight Control (male & female separately), data entry and evaluation.
- Army Physical Fitness Test (APFT), data entry and evaluation.
- Profiles, data entry.
- Profiles, custom query.
- Individual's weight history (report).
- Individual's APFT history (report).
- APFT, custom query.
- Weight Control, custom query.
- Physical Training, custom query.

(3) Dormitory Assignment

- Check person in.
- Check person out.
- Show assigned and available rooms.
- Show unassigned personnel.

b. Enter new service member into database (a(1) only)

- Display fields for data entry.
- Verify validity of data (SSN, dates, class name).
- List entry options (when feasible).

- Automate data entries (DOB -> age, ZIP -> city name).

c. Locate a service member in database (a(1) and a(2))

- Retrieve & display list of all personnel in unit (Company).
- Retrieve & display list of all personnel in sub-unit (platoon).
- Retrieve & display individual's records of specific category.

d. Enter/Modify data

- Indicate field to be filled/modified.
- Verify data validity (SSN, date format, class name).
- Allow entries for group of records whenever feasible.

e. Save changes to records

- Verify validity of data.
- Save changes to individual or multiple records.

f. Generate reports

- Retrieve & display data in pre-designed reports.
- Retrieve & display data in ad hoc query reports.
- Display list of data items available for query.
- Generate custom report.

g. Print reports

h. Export report data to a text file

5. Functional analysis

a. Display interface for record category

- Open window (new window for each group).
- Setup window for Admin/Bio.
- Setup window for Physical Training.
- Setup window for Dormitory Assignment.

b. Enter new service member into database (a(1) only)

- Setup Admin/Bio for new record.

c. Locate a service member in database (a(1) and a(2))

- Show personnel in unit/sub-unit.
- Retrieve data for selected/highlighted individual in currently active window.

d. Enter/Modify data - Verify validity of data.

- Navigate to the next field.

e. Save changes to records.

- Update database.

f. Generate reports

- Setup data-viewer window for specific report.

For ad hoc query:

- Open Query Builder window.
- Retrieve/display list of table columns.
- Build query from selected data items.
- Retrieve/display data.

g. Print reports

- Call PRINT() utility in active window.

h. Export report data to a text file

- Call EXPORT() utility in active window.

D. DATA COMMUNICATION

The current trend at POM is to connect every workstation to CAN. However, until this happens, some workstations need to connect to CAN via modem through telephone network, using terminal Server Access Controller System (TSACS). Such connections are not intended for continuous 24hr/day operation. Also, the typical database transaction executed via TSACS takes seconds or minutes, rather than milliseconds or seconds. Many MILDB transactions require retrieval of several reference data, before other queries can be formulated and executed. This would lead to significant time delays in execution of MILDB transactions, which would also make the use of the central database impractical. Rather, users without permanent connection to CAN should work on a local MILDB database, and regularly synchronize data with the central database.

III. KNOWLEDGE SYSTEMS

A. INTRODUCTION

Database is a collection of data serving some specific purpose. When applied in information system, this collection is usually also formally structured. In order to be considered a database, data don't have to reside in one location only. Rather, as it becomes typical today, data can be geographically distributed among several data repositories.

In order to be able to retrieve a specific data item from database, one can engage in devising a software that will find the physical location of data in data repository, read appropriate number of bytes, and format the output into a form, intelligible to the end user. A commercial database file system is an example of such approach. Then, according to some estimates, up to 80% of a typical business application is dedicated to coding a file access mechanism and to editing and validating the input data, while only about 20% of the application logic is dedicated to formatting and processing data before being displayed as output.

However, the mechanics of data storage and retrieval can be separated from database application and delegated to a separate software system, the Database Management System (DBMS). The database client application can then focus on business logic, data manipulation and presentation, while DBMS will store, modify, or extract data from database.

B. DBMS

1. Overview

Typical contemporary DBMS provides means for defining the type and layout of each data item (entity) to be stored in the database. These properties (attributes) can be referred to by a name (i.e., column name in a table). Data items (entities) are organized into larger wholes (tables, or relations). Various relationships and dependencies can be defined between tables (relations). Definitions of entities, definitions of relations, and relationships are parts of database schema, which is stored and maintained in a system catalog. There are several tasks to be accomplished during implementation of a database:

- Planning (define entities, relations, interdependencies, etc.).
- Construct a database (implement the plan).
- Populate the database with data (store initial data).
- Query the database (request, store new, or update existing data).
- Maintain the database (compact the database, add/remove indexes, etc).

To fulfil these tasks, we communicate with the DBMS using a language that DBMS understands. To create database objects (tables, indexes, and so on) we use Data Definition Language (DDL). In order to determine which values are present in a database at any given time, we use Data Manipulation Language (DML). Another language, Data Control language (DCL), is a set of commands that determine whether a user has appropriate permissions to perform a particular action. In reality, these languages are not separate. Rather, they are divisions of commands of a single

language, Structure Query Language (SQL). SQL standard is defined by ANSI (the American National Standard Institute). All commercial DBMSs have to conform to SQL-89 standard. It is desirable, however, that new DBMSs conform to a newer, SQL-92 standard. Various business applications and tool kits designed to interact with databases also provide a set of powerful commands that fulfil most of the tasks that SQL does. But they use simpler, more English-like languages, called fourth-generation languages (4GLs). Examples of such tool kits include PowerBuilder from Sybase, Visual Basic from Microsoft, Windows/4GL from Computer Associates, SQL-Forms from Oracle, and others. In this project, PowerBuilder 7 for Windows will be used.

2. Preferred DBMS Requirements

a. Interoperability

(1) Hardware

The DBMS should run on a variety of hardware platforms, fitted with one or more processors of no particular type. It should run on a single, off-line workstation, as well as in a multi-tier environment. It should also be capable of interoperating with legacy systems at both the data and application level.

(2) Database

Without requiring a separate installation procedure, the DBMS should provide the following functioning:

- *Heterogeneous data access* (select data from more then one database in a single query, either using native build-in mechanisms, or Open Database Connectivity interface).

- *Transaction Integration* (complete a query submitted using different kinds of query languages [SQL, Oql], or interface languages [HTML, Java]).

- *Replication* (ability of data modifications to non-native data stores, without the necessity of installing additional software).

- *Messaging System* (ability to remotely notify administrators about system errors, or other messages via, for example, e-mail).

b. DBMS Engine

- Automatic, Transparent Database Tuning

Based on usage patterns, reorganize data and index pages to improve performance, allocate additional memory space as needed, reclaim unused disk space in database files, verify integrity of data possibly compromised by hardware or software errors, cache the most frequently called stored procedures, verify data integrity by checking the structural integrity of data objects.

- Index Auto-Create

Based on usage patterns, optimize accessing data by creating/dropping indexes that are not part of the original design.

- Database Statistics

Automatically gather statistics about distribution of the data in the database for use by query optimizer, or database administrator.

- Dynamic Configuration

Continuously coordinate with the operating system a re-allocation of main memory, data and procedure cache.

- Locking

Based on the data amount, automatically determine the best locking strategy of records.

- Query Processor

Automatically optimize index maintenance, constraint checking, and parallel data load operations during import/export of large volume of data.

- Cursors

Enable locating/update of a record within a result set by supporting relative positioning indicated by graphical user interface.

- Table Design

Allow adding/removing columns regardless of data type, at any time and in any order, without loosing any data stored in database.

c. Administration

- Interface

Provide intuitive graphical interface for performing even elaborate administrative tasks.

- Scriptable Administration

Execute commands written in a variety of scripting languages, such as Java, Pearl, VB Script, etc.

- Multi-server Administration

Provide the ability to administer a multitude of subscribed servers from any workstation that has the administrative interface.

d. Data Movement

Integrate tools for data export/import from any type of data store. Transform/translate the data as it is moved from source to destination.

e. Data Warehousing

DBMS should include facility for creating, accessing, and manipulating a multidimensional database

f. Replication

DBMS should allow to replicate data to non-native data stores without the necessity to install additional software.

g. Tools

- Data Tools

Provide graphical tools for creation of database tables, queries, views, stored procedures, database diagrams, etc.

- Maintenance Tolls

Provide intuitive graphical interface for various administrative tasks, such as database creation, maintenance, performance tuning, replication setup, etc.

3. DBMS Classification

Range of data managed by DBMS may vary from simple to complex. Application that manipulates data may do so by means of queries, or without queries. In order to categorize few basic DBMSs, we will assume that data are either simple or complex, and application requires queries or does not require queries. Then we can describe four basic DBMS applications that manage:

a. Simple Data with Queries

A text editor is an example of a "no query" application. Text editor merely opens a file, and either overwrites the existing content with a new one, or appends the file content.

b. Simple Data without Queries

Simple data are those that can be expressed using standard data types found in SQL-89 or SQL-92. They can be captured in a two-dimensional table such as:

```
CREATE TABLE Sailors(
        Name        varchar(30),
        SSN         varchar(9),
        Rank        varchar(5),
        Unit        varchar(2),
        DOB         date,
        Weight      integer);

CREATE TABLE Units(
        Unit        varchar(2),
        CmdrSSN     varchar(9),
        Location    varchar(10));
```

To find all sailors serving at certain location (for example, POM), we can send the following query to the database:

```
SELECT name
        FROM Sailors
        WHERE Unit in (SELECT Unit
                        FROM Units
                        WHERE location = 'POM');
```

This type, or more complex, queries can be found in typical "business data processing" applications.

c. Complex Data without Queries

Typical application that falls into this category is CAD, where the computer must handle many interconnected items, some of which are complex themselves. Changes to any item could require extensive modifications to other items, in order for integrity of drawings to be maintained.

d. Complex Data with Queries

A digital library of pictures is a good example of unifying both complex items and queries. Each picture is scanned, and the location of each picture and location of selected features within the picture are recorded. User may query the database to find all pictures from specific area, or all pictures that contain certain feature.

For each of these case studies, a different DBMS is suited:

- Simple Data without Queries — File System
- Simple Data with Queries — Relational DBMS
- Complex data without Queries — Object-oriented DBMS
- Complex Data with Queries — Object-relational DBMS

All data in MILDB can be captured in 2-dimensional relations that will be queried. Relational DBMS (or, RDBMS) will be, therefore, studied and implemented.

4. Selection of RDBMS

Units will maintain their data either in a local database running on a workstation, or in the central database running on a network server. Each local database will hold data on several hundreds of personnel. The central database will hold data on personnel measured in thousands. Local MILDB database will hold approximately 5 MB of data, and can be managed by a small DBMS, such as Microsoft Access. However, more involved queries, such as:

```
DELETE FROM training
        WHERE ssn NOT IN (SELECT ssn
                          FROM admin);
```

can take minutes to complete even on a local database, which would be unacceptable for the central database, which will be queried simultaneously by many users. More powerful and effective database engine is needed for the central MILDB database. Microsoft SQL Server 7 was chosen to manage the central database. Brief description of selected DBMSs follows.

a. Microsoft Access 97

Microsoft Access can contain all its objects in a single file (.mdb). For this reason it is sometimes called a database container. Advantage of this fact is that the database file can be, when needed, easily transferred from one workstation to another and continue to function at the new location without any further special arrangements or interruption. The MDB file can also be easily backed up by making a simple copy to a different location, or can be forwarded to a different location for repair, in case the user is not experienced enough to perform such operation on site. Access 97 contains build-in

features for easy data import/export from external non-native data stores, as well as for linking external data sources to the database without actually importing the data. MS Access also provides access control features. Access control will not be implemented on a local MILDB database, since access to this database will already be controlled by the authentication procedure of the operating system.

System requirements for optimum performance:

Processor:	Pentium
RAM:	16~20MB (under Windows 95+)
	32 MB (under Windows NT 3.51+)
HARD Disk:	70 MB (for full installation of Access)
	10 MB (for database file)
Operating System:	Windows 95+
	Windows NT 3.51+

Hardware and software specifications of typical user's workstation meet, and exceed, the system requirements for MS Access 97.

b. Microsoft SQL Server 7

This product has evolved from DBMS developed by another relational DBMS vendor, Sybase. By today's standards, the SQL Server 7 meets many of the preferred requirements listed earlier in this secton. Its strengths include ease of use and, more significantly, its support of very large databases. SQL Server has build-in features such as dynamic self-management, high performance on-line backup, support of heterogeneous queries and English queries, data warehousing, data transformation

services, and others. SQL Server 7 is able to take advantage of multi-processor servers, and claims to be able to access directly up to 32 GB of memory by using 64-bit addressing. This by far exceeds the needs of the central MILDB database, which will in the future hold records (current and historic) measured in tens of thousands. However, SQL Server will be also utilized to manage other databases as well.

System requirements for optimum performance:

Processor:	Pentium 166 MHz +
RAM:	32 MB minimum
	(64 MB recommended)
HARD Disk:	70 MB (minimal installation)
	160 B (typical installation)
Operating System:	Windows 95+
	Windows NT 4.0+
	Service pack 4 or later
Other:	Internet Explorer 4.01+

Network server that meets and exceeds these requirements was procured.

C. DATABASE MODEL

1. Overview

From technical point of view, DBMSs can widely differ. Major types of DBMSs are: relational, network, flat, hierarchical, and object-oriented. Each type vary by the way the DBMS internally organizes information, which in turn can determine how quickly and

flexibly a user can extract information from database. In previous section we have concluded that relational DBMSs will be applied in this project. It is, therefore, the relational data model that will now be further investigated.

2. Relational Database Model

Relational database model has three main parts: structure, integrity, and data manipulation. Data structure defines the form for representing the data. Data integrity defines mechanisms for ensuring validity of stored data. Data manipulation provides means for manipulating data in database.

3. Data Structure

All information is stored and presented to users as two-dimensional relations (tables). Individual records (or tuples) in relations, equivalent to rows in tables, consist of fields (equiv. to columns in tables). The order of records in relations is immaterial. Each field (column) refers to an attribute. Attribute signifies the properties of the field, such as data type (char, integer, date), size, default value and, most significantly, the name of the field. This allows future references by database applications to this collection of attribute properties by a single name without knowing how a particular data item is stored in database.

Each tuple is a set of attribute-and-value pairs. The number of tuples in a relation is called cardinality. Since the ordering of tuples is immaterial, rows cannot be identified by row number. But each tuple can be uniquely identified by a set of attribute-and-value pairs, provided that no two tuples in a relation are the same. The minimum set of

attributes, whose values uniquely identify each tuple in a relation, is called a (candidate) key. If more than one candidate keys in a relation can be identified, one of them is arbitrarily chosen as the primary key of the relation.

Properties of relations can be summarized as follows:

- Each relation contains only one record type.
- Each relation has a fixed number of columns which are uniquely and explicitly named. Each attribute name within a relation is also unique.
- No two rows in a relation are identical.
- In each row, every attribute is atomic, that is, it has only one value set that cannot be further decomposed. Thus, no repeating groups are allowed.
- Ordering of rows is immaterial.
- Ordering of columns is immaterial.

4. Normal Forms

In theory, no two rows in a relation are identical. Also, within one row, no two columns are identical. This notion is easy enough to implement in a database having one or very few tables. This is rarely the case in a real-life database. By creating a multitude of tables with interrelated information, one can easily be fooled into believing that not only each single row in every relation is unique, but also that each attribute-value pair is unique simply by giving the attribute a different name in another (or event the same) relation. Then, the same information may be duplicated on several locations, which leads to wasted resources and, more importantly, to inconsistent data updates. Process of splitting relations with redundant information into two or more relations

without the redundancy is called normalization. A normal form is a way of classifying a table by its functional dependencies, which means: if I know the value of one attribute, I can always determine the value of another.

There are five main normal forms for relations:

- First Normal Form

Each attribute value is atomic. It cannot be a set, or other composite structure. By definition of data structure in relational database, each relation meets this criterion.

- Second Normal Form

Each relation is in at least the first normal form, and in addition, each non-key attribute depends on the entire primary key.

- Third Normal Form

Each relation is in at least the second normal form, and in addition, each non-key attribute depends only on the primary key.

- Fourth Normal Form

Each relation is at least in the third normal form, and in addition, there is no more than one multi-valued data item in the relation.

- Fifth Normal Form

A relation meeting the fifth normal criterion cannot be split into two or more tables (with each having its own primary key) without loss of information.

5. Data Integrity

Relational DBMS employs a mechanism which ensures that all data to be stored are valid. As a minimum, it needs to ensure that each attribute value is valid, check that the set of values in a tuple is unique, and that relations designed to be interrelated have, in fact, consistent values within each tuple that relates them.

a. Primary Key

Primary key is the only means of addressing a specific tuple within a relation. Therefore, in order to be unique, none of the primary key attributes can be null.

b. Domain

For some fields it may be useful to determine not only a data type, but also a range of permissible values. The following example demonstrates how declaring a domain can limit all possible values of "Location" to just a few, and ensure that valid entries for "Weight" are within permissible range:

```
CREATE TABLE Sailor(
        Ssn         char(9) NOT NULL UNIQUE
        Location    char(10) CHECK
                    (Location IN ('Monterey','Carmel','Pacific Grove','Seaside',
                    'Del Ray Oaks','Salinas')),
        Weight      integer CHECK
                    (Weight > 100 AND Weight < 200));
```

SQL-92 allows users to create domains as objects in a schema. Then, columns in tables can be declared as types of domain, rather than data types.

c. Foreign Key and Referential Integrity

When all of the valid values in one field of a relation (for example, Rank in Admin table) have to exist in a field of another relation (for example, Rank in ListOfRanks table), it can be said that the first field references the second and is, therefore, called a foreign key. The field to which it refers to is called its parent key. Names of the foreign and the parent key do not have to be the same. The parent key in the referenced table has to have either the UNIQUE or the PRIMARY KEY constraint in the table's definition, in order to ensure having unique values.

Some DBMSs implement referential triggered actions that have update effects and delete effects. These specify what happens when a parent key value in the referenced relation is modified or deleted. There are four options:

- SET NULL (sets to NULL all foreign keys that reference a parent key if it was modified or deleted).

- SET DEFAULT (same as above, but instead to set the referencing fields to null, foreign key values are changed to a preset default value).

- CASCADE (a change in the parent key value automatically triggers the same change in foreign key values).

- NO ACTION (the foreign value doesn't change, and if this would violate referential integrity, change of the parent key is disallowed).

6. Data Definition Language

Data Definition Language (DDL) commands allow to perform the following tasks:

- Create, alter, and drop databases and database objects.
- Grant and revoke access privileges and roles.
- Establish auditing options.
- Add comments to data dictionary.

DLL statements automatically update system catalog tables of DBMS.

7. Data Manipulation

As stated earlier, the Data Manipulation Language (DML) is not a language of its own. Rather, it is a subset of SQL. DML allows formulation of update queries and select queries. Here are four basic DML statements:

- **To add a tuple**

INSERT INTO <relation> VALUES <set of values> ;

- **To remove a tuple(s)**

DELETE FROM <relation>

WHERE <condition on attributes>;

- **To modify tuple(s)**

UPDATE SET <set of new values> IN <relation>

WHERE <condition on attributes>;

- **To select tuple(s) and join**

  SELECT <attributes>

  FROM <relation(s)>

  WHERE <selection criteria>;

Some aggregate functions that are used with SELECT:

COUNT, MIN, MAX, AVG, SUM, etc.

Set operations:

[NOT] IN, [NOT] EXISTS, CONTAINS (subset check), UNION, INTERSECT, MINUS.

8. Data Control Language

Data Control language (DCL) consists of statements that control security and concurrent access to data in relations. Common DCL commands are:

- **COMMIT** (instructs the DBMS to make permanent all data changes resulting from DML statement executed by a transaction).

- **CONNECT** (connects user to database).

- **GRANT** (assigns access privileges to a database user).

- **REVOKE** (revokes access privileges to database).

- **ROLLBACK** (reverses the effect of any DML command executed by a transaction, provided that backward log exists and is actively used).

- **LOCK / UNLOCK TABLE** (locks/unlocks a table from being accessed by other database users).

D. DATABASE SCHEMA

Database schema of a distributed database needs to be studied in two contexts:

- Design of individual databases.
- Integration of collection of databases into a global schema.

1. Local Database Schema

Database schema of individual databases contains logical description of data stored in a database. The schema defines the names of data items, their sizes and other attributes, and also identifies the relationships among the items.

Database schema of a local MILDB database is documented in Appendix B.

2. Global Database Schema

Local MILDB databases will be queried only by local users. Local users with proper privileges, local replicators, will replicate selected data items into the central database, where they will become available to global users. Local replicators will also replicate certain data from central database into their local MILDB, where it will become available to other local users. Data exchange will occur between tables in local MILDBs,

and views found in the central database. Figure 1 shows partial global database schema.

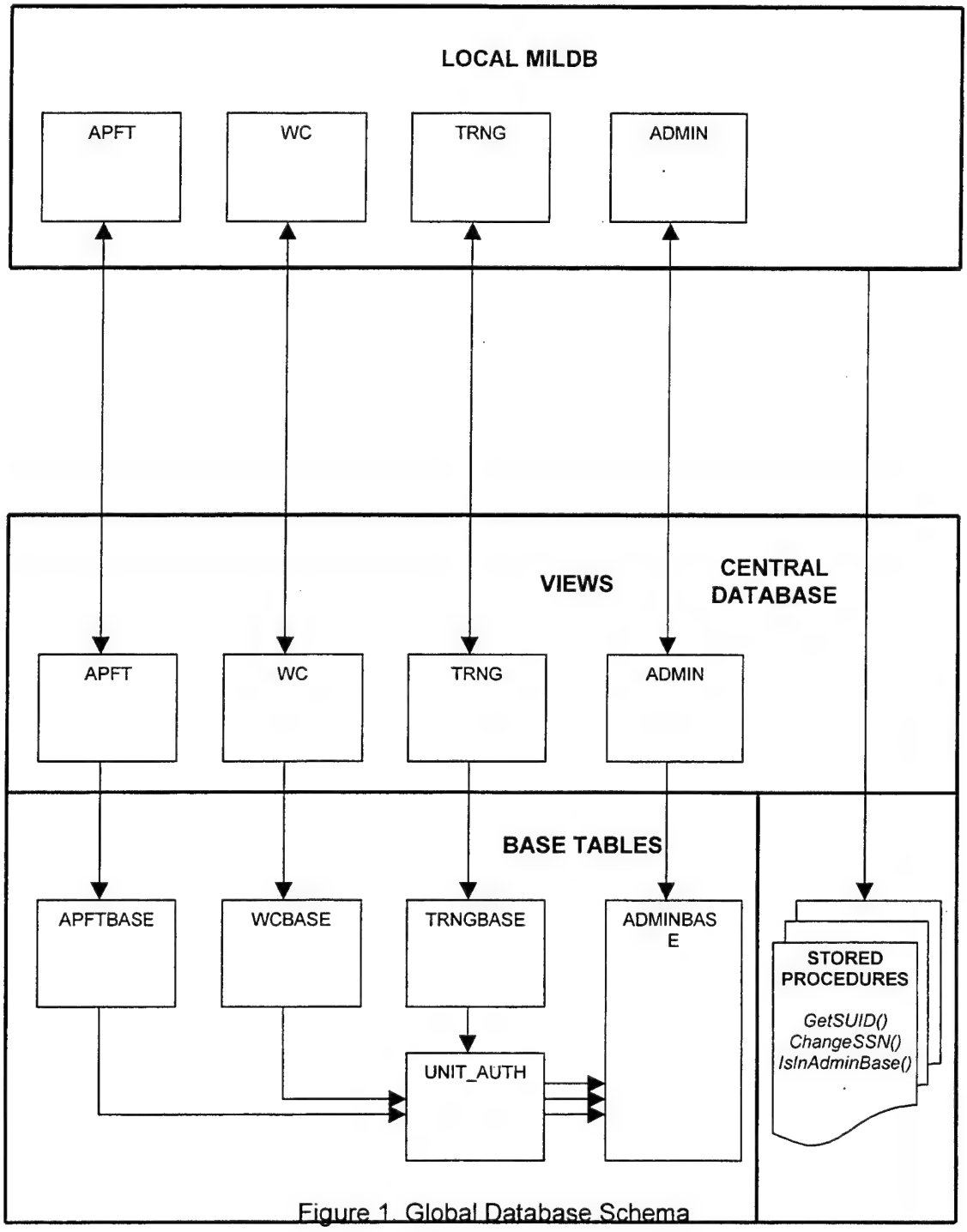


Figure 1. Global Database Schema

E. ACCESS CONTROL

The goal of this portion of thesis is to establish a firm access control to MILDB database and its data. Data are to be displayed to users on need-to-see basis. This means that after gaining access to MILDB database via login procedure, users will be allowed to view and manipulate only certain columns, and within the scope of these columns only those rows, that the user is authorized to see and manipulate.

1. SQL Access Control

As mentioned earlier, database access control is implemented by a set of DCL commands. Generally speaking, DBMS administrator (a user with the ultimate control over any database managed by DBMS) can create other users and give them certain privileges. A user, who creates a table, has control over this table (is the table's owner), and can in turn grand various privileges on this table to other users. Privileges are authorization identifiers that determine, whether or not a particular user can perform a given SQL command from DDL, DML, or DCL set of commands. Privileges are given and taken away by GRANT and REVOKE SQL commands. Two basic SQL statements can be used:

GRANT <privilege type> ON <object> TO <user id> ;

REVOKE <privilege type> ON <object> TO <user id> ;

a. SQL Access Control

As mentioned earlier, database access control is implemented by a set of DCL commands. Generally speaking, DBMS administrator (a user with the ultimate control over any database managed by DBMS) can create other users and give them certain privileges. A user, who creates a table, has control over this table (is the table's owner), and can in turn grand various privileges on this table to other users. Privileges are authorization identifiers that determine, whether or not a particular user can perform a given SQL command from DDL, DML, or DCL set of commands. Privileges are given and taken away by GRANT and REVOKE SQL commands. Two basic SQL statements can be used:

GRANT <privilege type> ON <object> TO <user id> ;

REVOKE <privilege type> ON <object> TO <user id> ;

To grant and maintain privileges for every individual database user separately would be too time consuming and could become too complex. Therefore, DBMSs implement a notion of *group* (or *role*, in SQL Server 7) to which privileges can be granted. Individual users can be added to, or removed from, the group and automatically inherit or lose a set of privileges granted to the group.

b. SQL Server 7 Access Control

Similar to principles implemented in other DBMSs, a user needs first to gain access to DBMS, after which he/she can perform operations on a database determined by the role he/she has been assigned. SQL Server 7 also introduces new special role, the *application role,* which will be studied later in this section.

c. Security Modes

To gain access to the SQL Server 7 engine, the user has to pass an authentication test. Two authentication methods can be implemented on the SQL Server engine in general, and on individual databases: Windows NT Server Authentication mode, and Mixed mode.

When Windows NT Server Authentication mode is applied, the engine checks, whether the Windows NT login ID of a user (who must have previously logged in on a Windows NT workstation) has been granted access to the database engine. Therefore, the user is not challenged by a separate login dialog box when he/she tries to connect to database via some interface, because his/her Windows login ID is automatically used.

When Mixed mode is applied, the user is challenged by a login dialog box. After login ID and password (PSW) is submitted, the engine tries first to authenticate the user via Windows NT authentication. If the login ID is not in Windows NT authentication database, the engine checks its internal user database.

d. Roles

Notion of *group* is in SQL Server 7 replaced by *role*. Groups are used in Windows NT operating system, where they have a function parallel to typical DBMS, but applying only to privileges related to the operating system. Entire NT groups can be assigned to SQL Server roles. Permissions granted, revoked, or denied to a role automatically apply to all users and groups assigned to "play" this role. Roles can be further nested.

Certain roles in SQL Server are predefined:

Fixed Server roles allow to perform various tasks on the SQL Server; *Fixed Database* roles allow to perform administrative tasks on individual databases.

User-Defined role is customized set of privileges that allows its "players" to perform various tasks on a single database. User-defined roles are local to each database.

Public Role provides a default (but customizable) set of permissions to a user who has not been assigned any other role.

Application Role allows to restrict access to data based on the application through which a user gained access to the database. Regardless of the role that the user might have been assigned to in the database, his/her role privileges are temporarily suspended for as long as he/she stays connected to the database via the application. Both security modes (Windows NT or Mixed) can be used by the application, but no "live" users can be associated with this role, since it is the application that has its own log ID and PSW. Typically, this log ID and PSW is hidden from users. Application role allows great flexibility to database administrators. It may, for example, allow less restrictive access to be granted to the application, since actions of its users will be controlled by the application, which can prevent them from performing malicious action or honest mistake. When connected through some other interface, the same users can have much more restrictive permissions (for example, read-only on certain tables).

e. Permissions

Login ID allows user to merely connect to SQL Server. A database user ID (same as login ID most of the time, but can be different), on the other hand, allows a user to access a specific database. But it is the set of *permissions* that allow a user to access

and manipulate objects within the database. Permissions can be assigned to individual users, or to roles and groups (meaning, Windows NT groups). SQL Server recognizes three different types of permissions: statement, object, and implied.

- **Statement Permissions**

Statement permissions allow users to execute commands typically found in DDL. Statement permissions include the following self-explanatory Transact-SQL (see further) statements: CREATE DATABASE, CREATE DEFAULT (creates default value), CREATE PROCEDURE, CREATE ROLE, CREATE TABLE, CREATE VIEW, BACKUP DATABASE, and BACKUP LOG.

Transact-SQL is Microsoft's version (dialect) of standard SQL-92. It is used for communicating between applications and SQL Server. A notable addition to standard SQL, found in Transact-SQL, are commands related to creating and manipulating stored procedures.

- **Object Permissions**

Object permissions include commands most commonly found in DML. They consist of the following Transact-SQL statements: DELETE, EXECUTE, INSERT, REFERENCE (ability to link tables), SELECT, UPDATE.

- **Implied Permissions**

Implied permissions are those that users automatically inherit just by the fact that they were assigned to a fixed server or database role, or because a particular user is an owner of a database object. Implied permissions cannot be assigned. Rather, a particular user needs to be included into a build-in fixed group that already has the permissions (i.e., database administrators).

f. Permission Precedence

Permissions can not only be granted and revoked, they can also be specifically denied. For example: a group of users has been assigned a role with a given set of permissions. We don't want one user from the group to have access to a certain view. Instead of removing that particular user from the group (because, possibly, it is more convenient to keep him in the group, so that he can perform some other chores assigned to this group in a different role), we can explicitly deny this user an access to that particular view.

Generally, the effect of permissions granted to a user is cumulative, with the exception of explicit denial of a permission. Denied access to an object overrules any permission the user would otherwise had by belonging to a group or role that enjoys that permission. If a user was denied access to an object in one group or role, he is automatically denied the same access in any other group or role, without the necessity to re-apply this denial in these other groups or roles. Permissions can be also denied to a group or role, for example, a trouble_makers role.

g. Ownership Chain

Whoever creates an object in a database, becomes the database object owner of the created object. If one user creates, for example, a table, then creates a view based on that table, and then creates a stored procedure that draws from that view, he/she is the owner of all three objects in the ownership chain, a *non-broken* ownership chain. If, on the other hand, user1 creates a table (user1 becomes the owner of this table), then user2 creates a view from that table (user2 owns the view), and yet another user, user3, creates a stored procedure that draws from the view (user3 owns the stored

procedure), such chain of ownership is in SQL Server 7 called a *broken ownership chain.* These two distinct cases have some implications: If the chain of ownership is non-broken, the owner of these objects needs to grant the permission only to the highest object in the chain (the stored procedure) in order to allow another user to operate on it. Appropriate permissions (SELECT, etc.) will automatically propagate down the chain. If, on the other hand, the chain of ownership is broken, in order to be able to grant permissions on the stored procedure, the owner3 of the stored procedure needs to obtain appropriate permissions from owner2 of the view, who in turn needs to obtain proper permissions from the owner of the base table, owner1. This situation is to be avoided. In order to prevent broken ownership chain, the fixed database role, the db_owner role, should be used. Anyone assigned to the db_owner role can create, manage, and manipulate any object within a database.

F. USING VIEWS FOR ACCESS CONTROL

Access to data in a database can be controlled by granting proper permissions on selected objects, such as tables and, more recently, even on individual columns. To maintain desired set of permissions on every table and even columns could become too time consuming complex. The same effect can be achieved in a much simpler way, by using views.

1. Overview

Views are virtual tables that display selected columns from one or more underlying tables. Only the definition of the view is permanently stored in the database. The view itself is constructed and populated with data only when it is called upon. Because views are virtual tables, they can be used the same way as the regular tables are: they can be used separately, or joined with another table or view; they can be used in unions; a view can also serve as a base table for another view. The only limitation is that not every view is updatable, as will be investigated in the next paragraph. Views are used extensively in the MILDB project.

Basic syntax for creating a view is:

CREATE VIEW <view name>

AS <query expression>

[WITH CHECK OPTION];

There are few restrictions on the SELECT clause in a view definition. It cannot:

- include ORDER BY, COMPUTE, or COMPUTE BY clauses.
- Include INTO keyword.
- Reference a temporary table.

Some DBMSs put more restrictions on view's definition.

Objects, referenced in view definition, must exist before the view can be created. In SQL Server, the SELECT clause can be of any complexity and can also include functions. If the optional WITH CHECK OPTION is specified, the view has to be

updatable. SQL Server also provides WITH ENCRYPTION option, which encrypts the CREATE VIEW statement in system tables. Also, in SQL Server, the definition of a view can be modified without having to drop and recreate the view, and therefore the permissions assigned to the original view are not lost and don't have to be re-established.

2. Updatable vs. Read - Only Views

In order for a view to be updatable, every row in the view has to be associated with exactly one row in the underlying base table. Naturally, a view which is based on a SELECT statement that contains, say, a UNION, does not meet this criterion and cannot be updated. View in SQL Server are updatable if:

- SELECT statement does not contain agregate functions (COUNT, MAX, etc). Agregate functions can be used, however, in a subquery in the FROM clause.
- SELECT statement does not contain GROUP BY, UNION, DISTINCT, or TOP.
- SELECT statement does not contain derived columns, such as those using functions, additions, or subtraction operators.
- FROM clause in the SELECT statement references at least one table.

A view of a view is updatable only if the source view is also updatable. UPDATE, INSERT, and DELETE statements can be executed only on a view that is updatable. UPDATE and INSERT statement must be written in such a way, that it modifies data in only one of the underlying base tables.

3. Using Access Control Table for Filtering Data in Views

As mentioned earlier, views display selected columns from one or more underlying base tables. If we have, for example, a base table named ADMINBASE that contains biographical and other data about students from all Units, and we want to restrict access to only those data that MILDB users need to see, we can define a view named ADMIN that will contain only selected columns from the ADMINBASE table. Such view will, after retrieval, contain selected data about all students in ADMINBASE. We want to further restrict the view to display only those rows that refer to students from a specific Unit, or even specific platoon within that Unit, based on user's access privileges. To achieve that, we can add another condition to the WHERE clause of the SELECT statement. Since the ADMINBASE table contains a Unit column, the view definition could look like this:

```
CREATE VIEW admin_a AS
        SELECT <column selection>
        FROM adminbase
        WHERE Unit = 'A' ;
```

Hard-coding the Unit's name into the view definition would, however, mandate creating separate views for every Unit, and the application would then be required to implement a mechanism that could select appropriate view based on user's privileges. That would not be practical. We need a single view that will be used by all Units, and still display only those rows that a particular user is allowed to see.

For this purpose, a small access authorization table was created. Itcontains user ID of each MILDB user, and Unit and Platoon of personnel that a particular user is

allowed to access. Wildcard character (%) can be used as an entry in the Platoon column to indicate access to data of all personnel in the Unit.

Authorization table definition:

```
CREATE TABLE mil_auth(
        n_user          char(8),
        unit            char(1),
        plt             char(1)) ;
```

Then we can construct a WHERE clause of the ADMIN view, that will include only those records from ADMINBASE whose Unit coincides with Unit(s) in table mil_auth for a particular n_user. But how can we identify the n_user in the WHERE clause? For this purpose we will invoke a special constant, built-in in most DBMSs, that has value USER. User is the authorization ID assigned automatically to any user after he/she/it logs on. Now we are ready to define view ADMIN as follows:

```
CREATE VIEW admin AS
        SELECT <column selection>
                FROM adminbase a
                WHERE a.unit + a.plt IN
                        (SELECT ua.unit + ua.plt
                                FROM unit_auth ua
                                WHERE ua.n_user = USER) ;
```

4. Implementation of Views

a. General

Every table found in local MILDB database has an equivalent table in the central SQL SERVER database, with a suffix "base". Thus, table ADMIN has its equivalent table ADMINBASE, APFT table has an equivalent table APFTBASE, and so on (with the exception of reference tables that have identical contents and, therefore, keep identical names). For each (non-reference) local MILDB table was in SQL Server database created a view having the same name and containing the same columns as tables in local MILDB, but with a WHERE clause similar to the one described earlier.

Summary of the system of MILDB tables and views:

LOCAL MILDB	CENTRAL MILDB	
Table	Table	View
ADMIN	ADMINBASE	ADMIN
APFT	APFTBASE	APFT
BRKS_ACT	BRKS_ACTBASE	BRKS_ACT
CC_TRNG	CC_TRNGBASE	CC_TRNG
Etc.	Etc.	Etc.

This arrangement has another add-on benefit: Views are virtual tables that, when properly defined, can be used just like the base tables. In fact, users may not be even aware of whether the object they are accessing is a base table or a view. This constitutes additional security mechanism for concealing parts of the database that may be confidential or are superfluous to a given user's needs. Giving the views in the central

database the names equivalent to tables in local databases gives the users impression that the central database contains tables just to meet the needs of the local MILDB.

b. Using Views in Joins

Joins are extensively used in MILDB database and in the interface application. Joins allow to retrieve data from two or more tables based on some logical relationship between the tables. Joins define how specific values retrieved from onetables are used to select the rows from another table by:

- specifying the column from each table to be used in the join;
- specifying a logical operator (=, <=, <>, etc.) to be used when comparing values from indicated columns.

Joins can be specified in the FROM or the WHERE clause. The basic SQL-92 syntax for a join in the FROM clause is:

```
SELECT <column selection>
        FROM table1 <join type> table2 ON <join condition> ;
```

Join type can be INNER, OUTER, or CROSS JOIN.

The *inner join* returns rows only when there is at least one row from both tables that matches the join condition. For example:

```
SELECT a.rank, a.name, c.task_num
        FROM admin a JOIN cc_trng c
                ON (a.ssn = c.ssn) ;
```

Rows from table ADMIN, whose SSN cannot be found in table CC_TRNG, are ignored.

The *outer join*, on the other hand, returns all rows from at least one of the tables, as long as those rows meet the condition specified in the WHERE or HAVING clause. If there is no matching row in the other table, an empty row is concatenated to the row(s) returned from the first table. For example:

```
SELECT a.rank, a,name, c.task_num
        FROM admin a LEFT OUTER JOIN cc_trng c
                ON (a.ssn = c.ssn) ;
```

This query will return all students from the ADMIN table. From the CC_TRNG table it will return all task_num values for each SSN that exists both in the ADMIN and the CC_TRNG table, and add it to the ADMIN portion of each row. For each SSN not found in the CC_TRNG table, a NULL (or default) value will be added to such row.

Cross join returns a Cartesian product of all rows found in both joined tables. As a consequence, it contains a lot of redundant data. It is not used in this project.

As described earlier, a set of views was created in the central database to act like virtual tables, equivalent to those found in local MILDB databases. Retrieving data from each of these views separately posed no problems. However, when two views were joined, we ran into some difficulties.

c. Restrictions

View ADMIN that contains biographical data provides basic information about each student, such as rank, name, ssn, etc. For this reason it is often joined with other tables (for example, a table containing training results) in order to give the data some meaningful identification to the end user.

The ADMIN view was defined as:

```
CREATE VIEW  admin as
        SELECT <column selection> FROM adminbase a
                WHERE EXISTS (SELECT * FROM unit_auth u
                                WHERE a.unit + a.plt
                                LIKE u.unit + u.plt AND u.n_user = USER) ;
```

Now, let's choose a table that contain some training data (i.e., cc_trng), and create a view that will restrict a user to seeing only rows pertinent to his/her Unit and Platoon. Five basic view definitions can be formulated:

A.
```
CREATE VIEW cc_trng AS
    SELECT <column selection>
        FROM cc_trngbase cb, adminbase ab, unit_auth u
        WHERE cb.ssn = ab.ssn
            AND ab.unit + ab.plt LIKE u.unit + u.plt
            AND u.user = USER ;
```

B. CREATE VIEW cc_trng AS

SELECT <column selection>

FROM cc_trngbase cb, admin a

WHERE cb.ssn = a.ssn ;

C. CREATE VIEW cc_trng AS

SELECT <column selection>

FROM cc_trngbase cb

WHERE EXISTS (SELECT *

FROM admin a

WHERE cb.ssn = a.ssn) ;

D. CREATE VIEW cc_trng AS

SELECT <column selection>

FROM cc_trngbase cb

WHERE EXISTS (SELECT *

FROM adminbase ab, unit_auth u

WHERE cb.ssn = ab.ssn

AND ab.unit + ab.plt LIKE ub.unit+u.plt

AND u.user = USER) ;

E. CREATE VIEW cc_trng AS

SELECT <column selection>

FROM cc_trngbase cb

WHERE EXISTS (SELECT *

FROM admin a

WHERE cb.ssn = a.ssn) ;

Each of these view definitions have compiled without error, and returned correct data when queried by statement:

SELECT * FROM cc_trng ;

Problems arose when the following queries were attempted:

SELECT a.rank, a.name

FROM admin a LEFT OUTER JOIN cc_trng c

ON (a.ssn = c.ssn) ;

SELECT a.rank, a.name

FROM admin a RIGHT OUTER JOIN cc_trng c

ON (a.ssn = c.ssn) ;

Results of these queries are summarized for each version of view CC_TRNG in the following table:

VIEW DEFINITION	RESULT	
	LEFT OUTER JOIN	RIGHT OUTER JOIN
A	Error message	Data returned
B	Error message	Data returned
C	Data returned	Data returned
D	Data returned	Data returned
E	Data returned	Data returned

Error message text: The table CC_TRNGBASE is an inner member of the outer-join clause. This is not allowed if the table also participates in a regular join clause.

d. Solution to Restrictions

SQL SERVER rejects a query in which a table is an inner member of the outer-join clause, and also participates in a regular join clause. The source of error becomes more apparent when we include the definition of view cc_trng in the SELECT query:

```
SELECT <column selection>
    FROM admin a LEFT OUTER JOIN ( SELECT <column selection>
                                   FROM cc_trngbase cb,
                                   adminbase ab, unit_auth u
                                   WHERE cb.ssn = ab.ssn
                                       AND ab.unit + ab.plt
                                       LIKE u.unit + u.plt
                                       AND u.user = USER)
                                ON (cc_trngbase.ssn=a.ssn);
```

When version A and B view cc_trng is applied, the table CC_TRNG participates, albeit indirectly via view definition, in the outer join of the FROM clause of the main query, and also in a regular join of the WHERE clause of the view definition. In versions C, D, and D of the CC_TRNG view, the table CC_TRNGBASE is also a member of a regular join, but in the subquery of the view's definition, which does not pose a problem.

Apparently, views do not always behave as regular tables after all. The syntax of their definition may preclude them from participating in joins with other tables or views.

e. Performance Issues

When used in a join with the ADMIN view, the three versions of the CC_TRNG view that returned data did not perform equally well. Performance comparison test was conducted for version C, D, and D of the CC_TRNG view during

a non-business day, when no other network traffic could affect the speed of data return. One version of the view was tested after another. Each set of tests was repeated five times. The test query:

```
SELECT   a.rank, c.ssn, c.task_num, c.task_no,
         c.score, c.d_tested, c,d_tran, c_n_user
    FROM admin a LEFT OUTER JOIN cc_trng c
         ON (a.ssn = c.ssn) ;
```

Test results are summarized in the following table:

Run	Version C		Version D		Version E	
	Time [ms]	Efficiency [ms/record]	Time [ms]	Efficiency [ms/record]	Time [ms]	Efficiency [ms/record]
1	44274	1.885	54519	2.321	41150	1.752
2	44254	1884	47348	2.016	43102	1.835
3	44414	1.891	47408	2.018	43573	1.855
4	44143	1.879	47408	2.018	41299	1.758
5	41910	1.784	44133	1.878	41299	1.758

Version E of the CC_TRNG view had consistently the best performance and was, therefore, implemented in the database. Good performance of this view can be apparently credited to the SELECT * clause in the WHERE EXISTS (SELECT * ...) statement, which lets the query optimizer of the DBMS decide which column to use. If some of the columns have indexes, the optimizer can use just these indexes to answer the query and never actually look at the table itself. The syntax of the version E was used in creating the rest of views that draw from base tables.

5. Stored Procedures

a. Overview

Stored procedures are sets of precompiled SQL statements that perform some operations on the database. Stored procedures, like tables and views, are objects. They are stored in the database, and require access permissions as any other object. Stored procedures, like views, allow to retrieve or modify information in sources to which the user would not normally have access. Such procedures are used in the MILDB project. Stored procedures can also be used to perform some database housekeeping chores and other operations on a database.

b. Implementation

Basic syntax for creating stored procedures:

```
CREATE PROCEDURE <procedure name>
    [<input_parm1 dataType>, <input_parm2 dataType>,
    <output_parm dataType OUTPUT>]
    [WITH RECOMPILE]
    AS
    <set of SQL statements> ;
```

The RECOMPILE option, used in SQL Server, forces the existing stored procedure to be recompiled if significant modifications were done to the original code.

Some stored procedures used in MILDB are simple, other are more involved. Example of a simple procedure is isInAdminBase(), which takes a Social

Security Number as input parameter, and returns record_status as OUTPUT. If a given SSN exists in ADMINBASE table, the procedure returns a letter indicating the status of the record, otherwise it returns "?":

```
CREATE PROCEDURE isInAdminbase(
        @newssn varchar(9), @recstat varchar(1) OUTPUT) AS

        SELECT @recstat = '?'
        SELECT @recstat = rec_stat
                FROM adminbase
                WHERE ssn = @newssn)
GO
```

Stored procedures can be nested. They can call other stored procedures or system function, create and delete temporary tables, and so on. Example of a more involved stored procedure, used in MILDB, is getSUID(), which takes two input parameters (name of a table, and User ID), and returns as output a combination of letters indicating set of basic permissions that the user has on a given table (S for SELECT, U for UPDATE, I for INSERT, and D for DELETE permission). If, for example, a user has only SELECT permission on a given table, the procedure will return "S???". This stored procedure also calls a system stored procedure USER_NAME:

```
CREATE PROCEDURE getSUID( @TABLE_NAME VARCHAR(384),
    @TABLE_USER VARCHAR(384), @TABLE_PERMS VARCHAR(4)  OUTPUT)
    AS
```

```
    if ( @TABLE_NAME is null) OR ( @TABLE_USER is null)
        begin
        raiserror 20001 'Must provide table name and user ID.'
        return
    end

DECLARE @sel char(1)
DECLARE @updt char(1)
DECLARE @insrt char(1)
DECLARE @dlt char(1)

SELECT @sel = '?'
SELECT @updt = '?'
SELECT @insrt = '?'
SELECT @dlt = '?'

SELECT @sel = 'S' FROM sysprotects p, sysobjects o, sysusers u, sysmembers m
    WHERE p.id = o.id
        AND o.type in ('U','V','S') AND object_name(o.id) = @TABLE_NAME
        AND user_name(u.uid) = @TABLE_USER
        AND (u.uid > 0 and u.uid < 16384)
        AND ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
        AND p.action = 193 /*select*/

SELECT @updt = 'U' FROM sysprotects p, sysobjects o, sysusers u, sysmembers m
    WHERE p.id = o.id
```

```
        AND o.type in ('U','V','S') AND object_name(o.id) = @TABLE_NAME
        AND user_name(u.uid) = @TABLE_USER
        AND (u.uid > 0 and u.uid < 16384)
        AND ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
        AND p.action = 197 /*update*/

SELECT @insrt = 'I' FROM sysprotects p, sysobjects o, sysusers u, sysmembers m
    WHERE p.id = o.id
        AND o.type in ('U','V','S') AND object_name(o.id) = @TABLE_NAME
        AND user_name(u.uid) = @TABLE_USER
        AND (u.uid > 0 and u.uid < 16384)
        AND ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
        AND p.action = 195 /*insert*/

SELECT @dlt = 'D' FROM sysprotects p, sysobjects o, sysusers u, sysmembers m
    WHERE p.id = o.id
        AND o.type in ('U','V','S') AND object_name(o.id) = @TABLE_NAME
        AND user_name(u.uid) = @TABLE_USER
        AND (u.uid > 0 and u.uid < 16384)
        AND ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
        AND p.action = 196 /*delete*/

SELECT @TABLE_PERMS = @sel + @updt + @insrt + @dlt
GO
```

The OUTPUT parameter value, returned when getSUID() procedure is called from MILDB application, is used to automatically enable/disable the 'Save' menu selection in the application's main menu.

Another example of a more complex stored procedure is changeViewSsn(), which takes as input parameters an old and new SSN. The procedure changes references to the old SSN in selected tables to a new SSN. To accomplish this task without violating data referential integrity, the procedure creates a temporary table #TEMPVIEWADMIN. This temporary table, which holds temporarily the admin data, is automatically dropped at the end of the procedure's execution. The definition of this stored procedure can be found in Appendix B.

c. Access Control

As mentioned earlier, stored procedures may allow access to objects which would otherwise be restricted to a particular user. This capability was utilized in the MILDB project. Here is a situation: a system of views, in conjunction with the authorization table, allow users to see personnel from only certain Unit, or even a platoon within that Unit. When the user needs to add a new service member into the database, new SSN must not violate the primary key on SSN in the ADMINBASE table. It is easy to check, whether new SSN exists within data viewable to the user, but how can the user verify the existence of SSN within records which he/she cannot see? Of course, the data integrity mechanism of DBMS would prevent the user from violating the primary key constraint in the ADMINBASE table and automatically trigger an error, whenever an insertion of a duplicate SSN were attempted. But we want to avoid undescriptive system-triggered messages, and we also want to be able to propose the

user some gracious alternatives. For example, activate a previously deactivated record. The stored procedure isInAdminbase(), described earlier, was designed specifically for this purpose. This stored procedure has access to the entire ADMINBASE table, and can verify the existence of any given SSN in it, while the user does not. The user is given only the EXECUTE permission on the stored procedure.

d. Executing Stored Procedures

Executing a stored procedure can be a one, or two step process. Database interfaces, non-native to a given DBMS, need to declare a stored procedure, before they can execute it. Here is an example of such stored procedure declaration:

```
DECLARE <alias_procedure_name> PROCEDURE FOR
        <procedure_name>
        [parm1 = value1, parm2 = value2,
        parm3 = value3 OUTPUT] ;
```

Then, the procedure can be executed by a statement:

```
EXECUTE <alias_procedure_name> ;
```

The same stored procedure can be executed from SQL Server native interface by a single statement:

```
EXECUTE <procedure_name> [<parm1, parm2, @parm3 OUTPUT] ;
```

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CLIENT SERVER ARCHITECTURE

Client server architecture emerged as a solution to the limitations of file sharing architectures. Using DBMS, client can query a database residing on server by means of SQL or remote procedure calls. Two basic types of arrangement of client/server architecture are possible: two-tier, and multi-tier model.

A. TWO-TIER MODEL

In two-tier model, the presentation and business logic of the application is deployed on the client computer, and the database management services, along with the database, are located on the server. This is a good solution when the number of users does not exceed 100 concurrent users. After that, the server may become burdened by keeping too many active connection open, and performance may start to suffer. Disadvantage of this model includes, besides the number of users limitation, necessity of complex data access control, heavy network traffic, and the necessity to deploy the entire application, along with supporting run-time DLLs, on every user's workstation.

B. MULTI-TIER MODEL

In this model, a middle tier is added between the user system and the database management server environment. The middle tier can serve as application server, message server, or processing monitor. In a typical three-tier model, for example, only the application's presentation logic is installed on client workstation, while the business

logic runs on the application server (the middle tier), and the DBMS with the database resides on another server. The three-tier architecture has demonstrated improved performances over the two-tier model, especially with a large number of users (in the thousands). Also, by centralizing the business logic on the application server, greater access control to sensitive information can be exercised.

C. SELECTION OF CLIENT SERVER MODEL

The MILDB database is going to be accessed by users from a single network domain. There will be far less than 100 users at any given time. Some MILDB users will be connected to the network only occasionally, but they will continue to access their data on their local MILDB databases, and will require the business logic of the MILDB application to be installed on their workstations.

Two-tier client/server model will be implemented.

V. DEVELOPING DATABASE APPLICATION IN POWERBUILDER 7

A. OVERVIEW

PowerBuilder is an object-oriented application development tool for building multitier applications that interact with databases. PowerBuilder applications consist of a user interface, and application processing logic. PowerBuilder applications are event driven. Users control application's behavior by the action they take. PowerBuilder provides a rich set of tools for accessing databases managed by variety of DBMSs. The application processing logic is formulated in a proprietary fourth-generation language, called PowerScript, which also permits queries written in SQL to be embedded in it. PowerBuilder for Windows is going to be applied in this project.

B. POWERBUILDER OBJECTS AND CONTROLS

Objects are the basic building blocks of any PowerBuilder application. They are:

- **Application**

Application object is the entry point into an application. It defines application-level behavior, such as what processes should occur when the application starts and closes. For example, if scripted accordingly, the open event of the application object will open the introductory window.

- **Window**

Window is the primary interface between the user and the application. A window consists of *properties*, that define the window's appearance and behavior, *events* triggered by user actions, and *controls* which are objects placed on a window (i.e., buttons, edit fields, text labels, dataWindow control, and others). Controls have a set of properties and events of their own.

- **DataWindow Object**

This object, fundamental to a typical PowerBuilder application, is used to retrieve and manipulate data from a database, or some other data store. This object also handles the way data is presented to the user. DataWindow object can not only contain data retrieved from database, it can also include computed fields that derive their values from the retrieved data. Pictures and graphs can also be tied directly to the retrieved data. DataWindow object does not display directly in a window. In order to be able to display it on the window, one has to create a DataWindow control, and then associate this control with desired DataWindow object.

- **Menu**

Menus are lists of items that a user can select from a menu bar assigned to the active window. Functionality of PowerBuilder's menus is equivalent to those found in other window applications.

- **Global Functions**

Global functions are self-contained pieces of programming code that can be called from any object within the PowerBuilder application. Other type of functions, *object-level* functions, can be called only within the scope of a particular object. Global and object-level functions can be user-defined, or build-in by PowerBuilder.

- **Queries**

This objet is a SQL statement, saved with a name identifier so that it can be used repeatedly anywhere in the application.

- **Structure**

Like in any other programming language, a PowerBuilder structure as a collection of variables of the same, or different, data types. Similar to functions, they can have global, or object-level scope.

- **User Object**

PowerBuilder has two types of user objects

Visual User Object (Reusable set of controls that has a consistent behavior. The Student Locator, found on most of the windows in MILDB application, is an example of such visual user object.)

Class User Object (Reusable non-visual user object serving as a processing module. A standard class user object of type transaction is used in MILDB application to declare and execute stored procedures.)

Custom Class User Objects (Non-visual objects that serve as building blocks in distributed PowerBuilder application. They provide various services, based on functions and variables defined in them.)

- **Library**

Library is a file (PBL file) in which PowerBuilder objects are saved. A PowerBuilder application can during its compilation draw objects from one or more PowerBuilder libraries.

- **Project**

Serves for creation of application executables and DLLs. Project object is used only in the developer's environment. It contains information about resource libraries, the type of executable, and other compilation options.

C. POWERSCRIPT LANGUAGE

1. Overview

PowerScript is a 4GL PowerBuilder language. PowerBuilder programming code, called script, consists of PowerScript commands, functions, and statements that are executed in response to events. Object-oriented capabilities of the PowerScript allow partitioning the business logic of an application into well-organized, reusable classes. PowerScript fully supports inheritance, encapsulation, and polymorphism.

2. Classes, Properties, and Methods

- **Classes**

Standard PowerScript classes include windows, menus, controls, and user classes. These are the foundation of visual objects. Non-visual objects are instantiations of standard class user objects (inherited from PowerBuilder system objects, such as Transaction, Message, or Error), or custom class user objects (inherited from PowerBuilder nonVisualObject class). PowerBuilder's nonVisualObject class allows to define an object class from scratch.

- **Properties**

Properties are defined by object variables and instance variables. Instance variables can be declared as public, protected, or private. This provides control how other objects' script can access them

- **Methods**

Methods include events and functions. A list of events, typical for any given object, is readily available for coding in the PowerBuilder programming interface. Additional events can be also included from PowerBuilder's own library of events, or from Windows events. PowerScript provides an extensive list of functions that can be used to act on various components of PowerBuilder application. Programmer can also declare and define his/her own functions in order to fulfil some specific task. Arguments can be passed to events and functions be value, by reference, or as read-only. In PowerBuilder 7, object events can be overriden in the chain of inheritance, and functions can be overloaded.

3. Global Variables and Functions

PowerBuilder implements several build-in global variables and functions. A programmer can also declare his/her own variables and functions which have global scope, and can be accessed from any script within the application.

4. Garbage Collection

The PowerBuilder garbage collection mechanism automatically checks memory, and destroys any unreferenced or orphaned objects.

D. COMMUNICATING WITH DBMS

Most of the communication with DBMS takes place via methods which are built-in PowerBuilder objects. For example, when user invokes an update() method of DataWindow, PowerBuilder generates and submits to DBMS all necessary SQL statements. But programmer can also formulate his/her own SQL statements, and embed them in PowerScript code. Embedded SQL statements have to be concluded with a semicolon (;). All SQL statements, embedded in scripts or dynamically generated by PowerBuilder, are executed by means of a transaction object.

1. Transaction Object

Transaction object is a special non-visual object that serves as an intermediary between PowerScript and the DBMS. The transaction object contains parameters that

PowerBuilder application uses to connect to a database. Every PowerBuilder application automatically creates a global default transaction object, named SQLCA (SQL Communication Area). This, or another transaction object, explicitly declared and created by script, can be used to connect, communicate with, or disconnect from database.

Before a connection can take place, at least some transaction object properties have to be set:

- DBMS (Indicates the DBMS that manages the database to be connected. Can be set to "ODBC".)

- Database (Name of the database to which the application will connect. Can be the name of ODBC data source (DSN), established in the ODBC interface.)

- DBParm (Contains DBMS-specific connection parameters, such as AutoCommit, Lock, DSN, ServerName, etc.)

Other parameters may be needed in order to establish a connection with database, such as Log ID, PSW, server name, and others, but they may be provided by DSN, or may be requested by the DBMS dialog box at the time of connection.
Connection to database is requested by statement:

CONNECT [using <transaction_object>] ;

If no transaction object is specified in the statement, the SQLCA is used by default. Other transaction management statements are:

- COMMIT; (makes permanent all changes made to the database)
- ROLLBACK; (all modifications to the database, performed by current transaction, are undone)
- DISCONNECT; (disconnects from database)

Application can connect to more than one database at a time. Each connection needs to be managed by a separate transaction object. Such arrangement is applied in MILDB, when data are synchronized between local MILDB and the central database.

Example of establishing a multiple database connection:

CONNECT using <sourceTransObj> ;

CONNECT using <destinationTransObject> ;

2. Transaction Object and Stored Procedures

Stored procedures can be executed by SQL statement, embedded in the script. This, however, will not work if the stored procedure returns an OUTPUT parameter, or when a stored procedure which contains a DLL statement is called. For this purpose, a customized version of the standard class user object of type transaction has to be used. Once such user object is created by means of PowerBuilder Object Wizard, stored procedures can be declared as external functions, or external subroutines, for that user

object. If the stored procedure has a return value, it must be declared as a function. If the stored procedure returns nothing or void, it must be declared as a subroutine. In both cases, a RPCFUNC keyword has to be used in the declaration.

Examples of stored procedures declared in a user object, applied in MILDB:

Subroutine getSUID(string table_name, string table_user, ref string table_perms)
RPCFUNC alias for "mil.getsuid"

Function int sp_setapprole(string roleanme, string password, script encrypt)
RPCFUNC alias for "dbo.sp_setapprole"

The second declaration is for a built-in SQL Server stored procedure which activates permissions associated with an application role. This stored procedure contains a DLL statement, and must be executed outside the scope of a transaction. To achieve that, this procedure must not only be declared as a function in a customized transaction object, but the autoCommit property of the transaction object must be set to TRUE before the procedure is called.

Once the stored procedures are declared in a customized transaction object, they can be executed from the application script by statement:

<trans_object_name>.<procedure_alias_name([parm1, parm2, ...] ;

3. DBMS Interfaces

PowerBuilder application can connect to a database through a standard database interface (i.e., ODBC, JDBC, or OLE DB), or through a native database

interface. A standard database interface communicates with the database via standard-compliant driver (ODBC, or JDBC connection), or data provider (OLE DB connection). The standard-compliant driver or data provider translate abstract function calls, defined in standard API, into calls understood by a specific DBMS. A native database interface communicates with DBMS via a direct connection to database, using a native API library.

a. ODBC Interface

Open Database Connectivity (ODBC) is a standard application programming interface (developed by Microsoft, is API which allows an application to access a variety of DBMSs. An ODBC-compliant driver, appropriate for a given DBMS, has to be installed on user's workstation. SQL is used for communication with the database. The ODBC specifies:

- A library of ODBC function calls for connecting to the database, executing SQL statements, and retrieving results.
- A standard way to connect and log on to a DBMS.
- SQL syntax.
- Standard representation for data types.
- Standard set of error codes.

PowerBuilder provides a set of ODBC drivers for the most common DBMSs, such as Sybase, Oracle, SQL Server, and others. ODBC drivers can be also obtained directly from DBMS vendor. Since the ODBC driver for Microsoft Access is readily

available on user's workstation, and only the SQL Server driver will have to be installed, the ODBC interface will be used in this project for communicating with databases.

When accessing an ODBC data source from a PowerBuilder application, the connection goes through several layers, before reaching the data source:

- Application (calls ODBC functions).
- ODBC Driver Manager (installs, loads, and unloads drivers for the application).
- Driver (processes ODBC function calls).
- Data Source (stores and manages data in a database).

b. OLE DB Interface

OLE DB, a component of Microsoft's Data Access Component software, is a standard API developed by Microsoft. It allows an application to access a variety of data for which OLE DB data provider exists. Data can be stored in a variety of forms: indexed-sequential files, spreadsheets, e-mail, personal databases, or full-fledged DBMS. An OLE DB data provider is a dynamic link library (DLL) that implements function calls. An application invokes the OLE DB data provider to access a particular data source. Some OLE DB providers are shipped with PowerBuilder, other can be obtained directly from a data source vendor.

c. Native Database Interface

A native database provides a direct native connection to a particular DBMS. It implements its own interface DLL, which communicates with the specific database through a vendor-specific API.

E. BUILDING THE MILDB APPLICATION

1. Application Architecture

The MILDB application consists of one application object, and multitude of dataWindow objects, global functions, menu objects, user objects, pipeline objects, and windows. All these objects are stored in a single PowerBuilder library file, named mil_0799.pbl. The architecture of the MILDB application is shown on Figure 2. The list of objects in the application is extensive. Some objects serve only one task, such as dataWindow objects that retrieve and display canned reports. Other objects, such as user objects or menus, are used repeatedly throughout the application. Even though every object shown in Figure 2 was developed and is fully functional, documenting each of them would exceed the scope of this thesis. Only several major components of MILDB application are documented in order to demonstrate how the application was developed, and how it functions.

2. Application Object

The application object, named Millshell, serves as the entry point into MILDB application. Through this object, global variables are declared and initialized, database

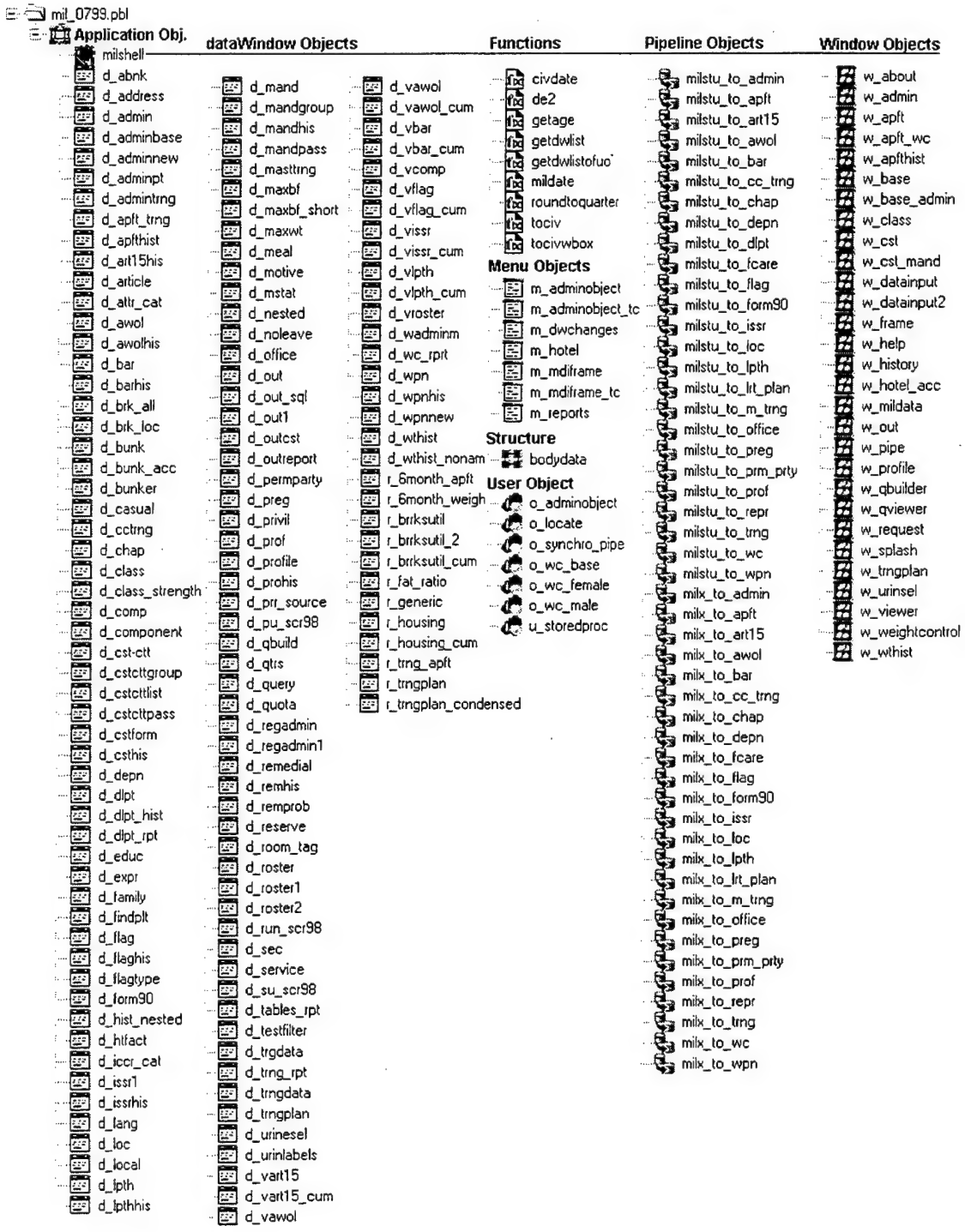


Figure 2. MILDB Application Architecture

connection is established, and the introductory window is opened. Application object is non-visual. The following code shows the script of the Millshell application object:

```
//***APPLICATION OBJECT

//Declare global external function
function int getwindowsdirectory(ref string buff, int sz)
            LIBRARY "kernel"


//Global variables
string globAppname       //application name
string globVersion       //application version
string fetch_ssn         //student SSN
string company           //Unit indicator
string expPath           //physical path of export file



//APPLICATION OPEN EVENT
//***Set GLOBAL VARIABLES ========================================

//**Company
//set in script of w_frame
//put "" (empty) to install troop command menu/toolbar
company = ""

//**Export path
expPath = "C:\MILX\export.txt"

//**Application Name & Version
globAppname = "MILdB - Military Database"
globVersion = "7.0" + company
//============================================================

//open splash windows
open (w_splash)

//setup ODBC
sqlca.dbms = "ODBC"
sqlca.DbParm = "ConnectString='DSN=milstu'"
SQLCA.AutoCommit = True


//***CONNECTION
//Try the central MILSTU as default first
connect;

//when connection fails, try connection to local MILDB
if sqlca.sqlcode < 0 then
   //Try connection to local datbase MILX
```

```
   SQLCA.Database = "mildb"
   sqlca.DbParm = "ConnectString='DSN=MILX'; "&
                  + "Time=' ''''hh:mm:ss:'''' ';"&
                  + "delimitidentifier='NO';"
   connect;
else
   //get logID of current user
   select distinct n_user into :sqlca.userID
      from mil.unit_auth
      where n_user = user;
end if

if sqlca.sqlcode < 0 then
   messageBox("DATABASE CONNECT","Could not connect to database")
   HALT
end if


//open MDI frame
open(w_frame)

//close splash window
close(w_splash)
```

3. DataWindow Objects

DataWindow objects are used to retrieve and manipulate data from database. Individual data fields (columns) of dataWindows can be setup as updatable, or read-only. Values, retrieved from database, can processed be further in dataWindow's calculated fields. Data can be displayed in various formats, and be arranged within a dataWindow as needed. Figure 3 through Figure 6 show examples of dataWindow objects used for retrieving, displaying, and manipulating student data. SQL statements for populating each dataWindow is also included.

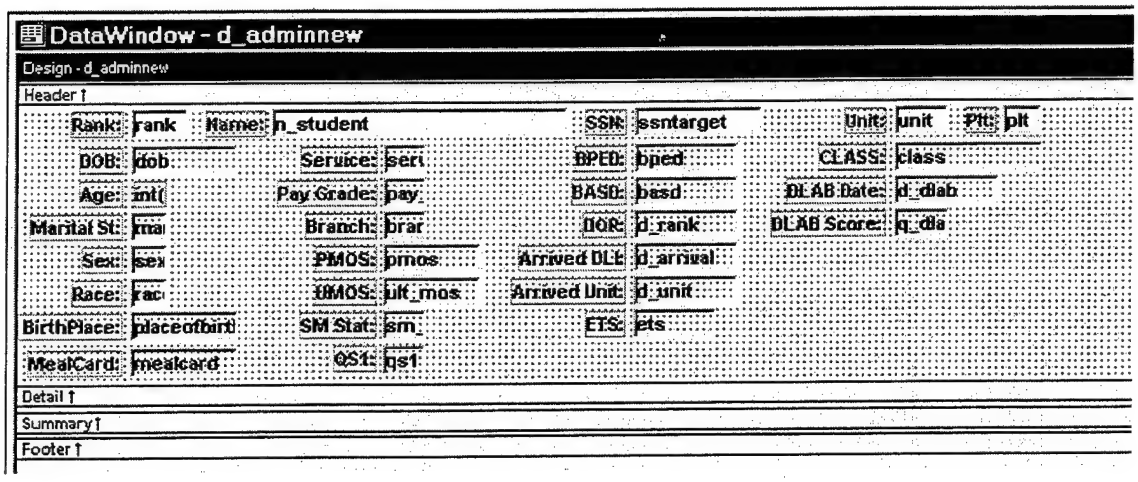


Figure 3. DataWindow d_adminnew

SQL statement for d_adminnew:

```
SELECT mil.admin.dob ,
   mil.admin.d_arrival ,
   mil.admin.d_rank ,
   mil.admin.pay_grd ,
   mil.admin.race ,
   mil.admin.ssn ,
   mil.admin.d_tran ,
   mil.admin.n_user ,
   mil.admin.bped ,
   mil.admin.basd ,
   mil.admin.ets ,
   mil.admin.d_unit ,
   mil.admin.mar_stat ,
   mil.admin.mealcard ,
   mil.admin.service ,
   mil.admin.branch ,
   mil.admin.pay_code ,
   mil.admin.d_dlab ,
   mil.admin.q_dlab ,
   mil.admin.pmos ,
   mil.admin.ult_mos ,
   mil.admin.sm_status ,
   mil.admin.lic_1 ,
   mil.admin.lic_2 ,
   mil.admin.qs1 ,
   mil.admin.sex ,
```

```
    mil.admin.unit ,
    mil.admin.n_student ,
    mil.admin.rank ,
    mil.admin.q_age ,
    mil.admin.plt ,
    mil.admin.class ,
    mil.admin.d_depart,
    mil.admin.placeofbirth
FROM mil.admin
WHERE ( mil.admin.ssn = :ssn )
```

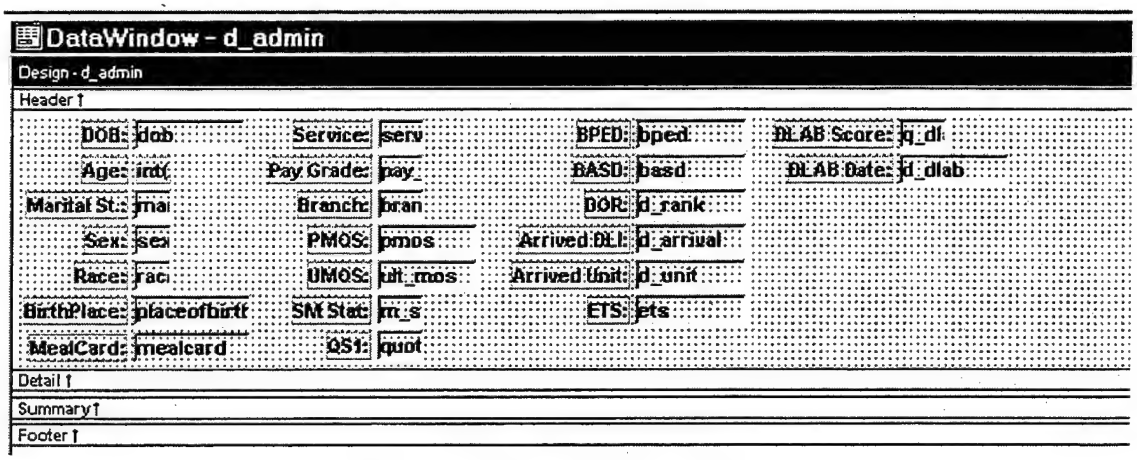


Figure 4. DataWindow w_admin

SQL statement for dataWindow d_admin:

```
SELECT mil.admin.dob,
   mil.admin.d_arrival,
   mil.admin.d_rank,
   mil.admin.pay_grd,
   mil.admin.race,
   mil.admin.ssn,
   mil.admin.d_tran,
   mil.admin.n_user,
   mil.admin.bped,
   mil.admin.basd,
   mil.admin.ets,
   mil.admin.d_unit,
   mil.admin.mar_stat,
   mil.admin.mealcard,
   mil.admin.service,
   mil.admin.branch,
   mil.admin.pay_code,
   mil.admin.d_dlab,
```

```
    mil.admin.q_dlab,
    mil.admin.pmos,
    mil.admin.ult_mos,
    mil.admin.sm_status,
    mil.admin.lic_1,
    mil.admin.lic_2,
    mil.admin.qs1,
    mil.admin.sex,
    mil.admin.unit,
    mil.admin.q_age,
    mil.admin.placeofbirth
FROM mil.admin
WHERE (mil.admin.ssn = :ssn)
```

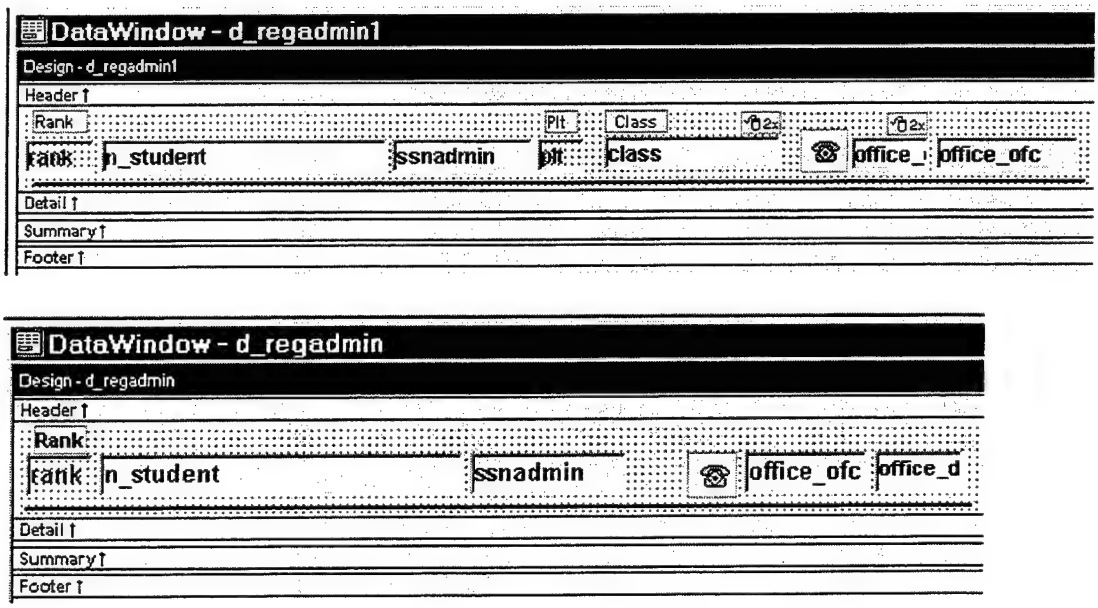


Figure 5. DataWindow d_regadmin1 and d_regadmin

SQL statement for d_regadmin1 and d_regadmin:

```
SELECT mil.admin.class,
    mil.admin.dob,
    mil.admin.d_arrival,
    mil.admin.d_rank,
    mil.admin.n_student,
    mil.admin.pay_grd,
    mil.admin.plt,
    mil.admin.race,
    mil.admin.rank,
    mil.admin.sex,
    mil.admin.ssn,
    mil.admin.d_tran,
```

```
    mil.admin.n_user,
    mil.office.ofc,
    mil.office.dty_phon
FROM {oj mil.admin  LEFT OUTER JOIN mil.office
    ON mil.admin.ssn = mil.office.ssn}
WHERE mil.admin.ssn = :ssn
```

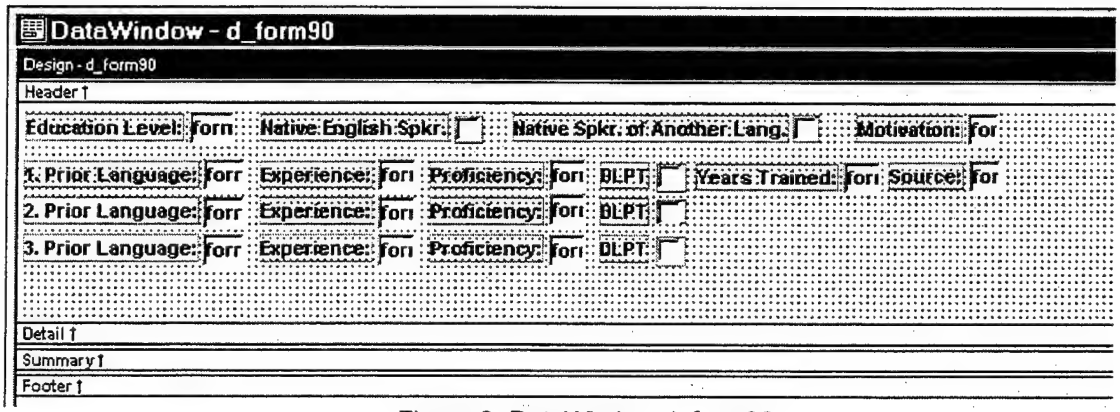


Figure 6. DataWindow d_form90

SQL statement for d_form90:

```
SELECT mil.form90.educ_lvl,
    mil.form90.yrs_svc,
    mil.form90.motivate,
    mil.form90.natv_eng,
    mil.form90.natv_oth,
    mil.form90.prr_lang,
    mil.form90.yr_trained,
    mil.form90.prr_prof,
    mil.form90.source,
    mil.form90.prr_dlpt,
    mil.form90.prr_expr,
    mil.form90.ssn,
    mil.form90.d_tran,
    mil.form90.n_user,
    mil.form90.prr_lang2,
    mil.form90.prr_lang3,
    mil.form90.prr_prof2,
    mil.form90.prr_prof3,
    mil.form90.prr_expr2,
    mil.form90.prr_expr3,
    mil.form90.prr_dlpt2,
    mil.form90.prr_dlpt3
  FROM mil.form90
 WHERE ( ( mil.form90.ssn = :ssn ) )
```

4. Global Functions

There are several global functions designed to perform certain operations repeatedly throughout the application. The following code shows the declaration, purpose, and script of MILDB global functions:

```
//***GLOBAL FUNCTIONS

//PARAMETERS:  string dateText   - military date
//RETURNS:     1 if success, or -1 if invalid date
//PURPOSE:     converts & modifies referenced date
//             from military format to civilian
integer  civdate ( ref string datetext )

//accepts reference to string containing military date
//converts & modifies referenced date from military format to civilian
//Civilian date format: MM/DD/YY
//Military date format: YYMMDD
//returns 1 if success, or -1 if invalid date

//parameter: date dateText

//Local variables
date milDate          //date in military format

mildate = date(dateText)

if year(milDate) = 1900 then

   string newDate, yr , mo, dy

   newDate = dateText
   yr = left(newDate, 2)
   mo = mid(newDate, 3, 2)
   dy = mid(newDate, 5)

   if isNumber(yr) then
      if integer(yr) >= 50 then
         yr = "20" + yr
      else
         yr = "19" + yr
      end if

      newDate = mo + "/" + dy + "/" + yr
      milDate = date(newDate)

      if year(milDate) = 1900 then
         messageBox("DATE", &
```

```
         "Enter date in one of the following formats: " &
         + "YYMMDD , or MM/DD/YY, or other format used by Windows.", &
         Exclamation!)

         return -1
      else
         dateText = newDate
      end if
   else
      messageBox("DATE", "Enter date in one of the following formats: "
&
         + "YYMMDD , or MM/DD/YY, or other format used by Windows.", &
         Exclamation!)
      return -1
   end if
end if

return 1


//PARAMETERS:  string decIn    - decimal number in string format
//RETURNS:     string str      - decimal number in string format
//PURPOSE:     formats a decimal number to contain at least
//             one leading, and two trailing zeros
string  de2 ( string decin )

//Local variables
string str  //formated decimal string

str = right("    " + string(dec(decin), "##0.00"),6)
return str



//PARAMETERS:  date dob        - date of birth
//             date ageDate    - date when age is calculated
//RETURNS:     integer         - age in years
//PURPOSE:     formats a decimal number to contain at least
integer getage ( date dob, date agedate )

return ( daysAfter( toCiv( dob), toCiv( ageDate))/365)


//PARAMETERS:  window sourceWindow  - owner of dataWindows
//             datawindow dwArry[]  - array of dataWindows
//RETURNS:     integer              - number of dataWindows
//PURPOSE:     Counts and returns the number of
//             all dataWindows on the referenced window
integer getdwlist( readonly window sourcewindow, ref datawindow
dwarry[] )

//Finds all dataWindow controls in Window
//and puts references to them into array passed as argument
//returns number of dataWindow controls
```

```
//Local variables
dataWindow dwList[]  //temp array of dataWindows

int dwCnt            //dataWindow counter
int ii               //step counter

//initialize local variable
dwCnt = 0

//browse through all objects in the window
//when the object is dataWindow, put it in dwList[]
for ii = 1 to upperBound( sourceWindow.control[])
   if sourceWindow.control[ii].typeOf() = dataWindow! then
      dwCnt++
      dwList[dwCnt] = sourceWindow.control[ii]
   end if
next

//assign temp arry to referenced dataWindow array
dwArry = dwList

return dwCnt


//PARAMETERS:  userobject sourceobj - owner of dataWindows
//             datawindow dwArry[]  - array of dataWindows
//RETURNS:     integer              - number of dataWindows
//PURPOSE:     Counts and returns the number of
//             all dataWindows on the referenced user object
int getdwlistofuo(readonly userobject sourceobj, ref datawindow
dwarry[] )

//Finds all dataWindow controls in source userObject
//and puts references to them into array passed as argument
//returns number of dataWindow controls

//Local variables
dataWindow dwList[]  //temp array of dataWindows

int dwCnt            //dataWindow counter
int ii               //step counter

dwCnt = 0

//browse through all objects in the user object
//when the object is dataWindow, put it in dwList[]
for ii = 1 to upperBound( sourceObj.control[])
   if sourceObj.control[ii].typeOf() = dataWindow! then
      dwCnt++
      dwList[dwCnt] = sourceObj.control[ii]
   end if
next
```

```
//assign temp arry to referenced dataWindow array
dwArry = dwList

return dwCnt


//PARAMETERS:  date civDate      - date in civilian format
//RETURN:      string milDate    - date in military format
//PURPOSE:     converts civDate to string containing
//             military date format of civDate
string mildate( date civDate)

//takes argument: date civDate, passed by value
//converts civDate to string containing military date format of civDate
//returns string milDate

string milDate    //date in miltary format

if year( civDate) = 2000 then
   milDate = string( civDate, "YYYYMMDD")
else
   milDate = string( civDate, "YYMMDD")
end if

return milDate



//PARAMETERS:  decimal  xVal     - decimal number
//RETURN:
//PURPOSE:
dec roundToQuarter( decimal xVal)

//Receives parameter DECIMAL xVal
//rounds passed value to the nearest quarter

//Local variable
dec{2} roundVal

roundVal = (int(xVal / 0.25))*0.25

if (xVal - roundVal) >= 0.125 then
   roundVal += 0.25
end if

return roundVal


//PARAMETERS:  any milDate    - date in variable of any data type
//RETURN:      any ( or NULL value if empty parm submited)
//PURPOSE:     Converts date or string into
//             civilian date format mm/dd/yyyy
any toCiv( any mildate )
```

```
//If parameter is date,   function returns date (in variable of type
any)
//If parameter is string, function returns string

//Local variables
string civDateStrng     //civilian date string
string parmType         //parameter type
string yr, mo, dy       //year, month, day

date civDate            //civilian date

//get parameter type
parmType = ClassName( milDate)

//cast the date
civDateStrng = trim( string( milDate))

if civDateStrng = "" then
   setNull( civDateStrng)
   setNull( civDate)
else
   if isDate( civDateStrng) then

      civDate =  date( string( mildate, "mm/dd/yyyy"))

   else
      //two right-most digits => day
      dy = right( civDateStrng, 2)

      //first two of the four right-most digits => month
      mo = right( civDateStrng, 4)
      mo = left( mo, 2)

      //remaining digits,
      //after disregading four right-most digits, => year
      yr = left( civDateStrng, ( len( civDateStrng) - 4))

      civDateStrng = mo + "/" + dy + "/" + yr

      if isDate( civDateStrng) then
      civdate = date( civDateStrng)
      end if
   end if
end if

if parmType = "string" then
   return civDateStrng
end if

return civDate
```

```
//PARAMETERS:  any milDate    - date in variable of any data type
//             any actionparm - indicates option chosen
//                              by user in messageBox
//RETURN:      any ( or NULL value if empty parm submited)
//PURPOSE:     Converts date or string into
//             civilian date format mm/dd/yyyy
any toCivwBox ( any mildate, ref any actionparm )

//If parameter is date,   function returns date (in varuable of type
any)
//If parameter is string, function returns string

//Local variables
string civDateStrng     //civilian date string
string parmType         //parameter type
string yr, mo, dy       //year, month, day

int boxRtrn             //indicates option chosen by user in messageBox

date civDate            //civilian date

//get the type of parameter
parmType = ClassName( milDate)

civDateStrng = string( milDate)

//initialize the message to be displayed
msg = "Enter date in one of the following formats:~n~n"&
      + "YYYYMMDD, or YYMMDD, or MM/DD/YYYY, or MM/DD/YY.~n"&
      + "Use leading 0 (zero) for days and months below 10 when"&
      + " using the military date format.~n~n"&
      + "EXAMPLES: 20010430, 010430, 4/30/2001, 4/30/1~n~n"&
      + "If the year is displayed as two digits, the century is "&
      + "determined as follows:~n"&
      + "Year is between ~tDefault Century Digits   ~tEXAMPLE~n"&
      + "00 and 49       ~t20~t~t~t2049~n"&
      + "50 and 99       ~t19~t~t~t1976~n~n"&
      + "Include the century (e.g. 3/27/1944, 19440327) when you "&
      + "want to override "&
      + "the default interpretation of a two-digit year, "&
      + "or not certain how the date will be interpreted by the
program."


if isDate( civDateStrng) then

   civDate =  date( string( mildate, "mm/dd/yyyy"))
   setNull( msg)

else
   //two right-most digits => day
   dy = right( civDateStrng, 2)

   //first two of the four right-most digits => month
```

```
   mo = right( civDateStrng, 4)
   mo = left( mo, 2)

   //remaining digits, after disregading four right-most digits, =>
year
   yr = left( civDateStrng, ( len( civDateStrng) - 4))

   civDateStrng = mo + "/" + dy + "/" + yr

   if isDate( civDateStrng) then
      civdate = date( civDateStrng)
      setNull( msg)
   end if
end if

if isNull( msg) then

   //indicate conversion success
   boxRtrn = 0

else
   choose case lower( string( actionParm))

      case "retrycancel"
         //indicate user selection in dialog box
         boxRtrn = messageBox( "DATE", msg, exclamation!, retryCancel!)

      case else
         //indicate user selection in dialog box
         boxRtrn = messageBox( "DATE", msg, exclamation!)

   end choose
end if

actionParm = string( boxRtrn)

if parmType = "string" then
   return civDateStrng
end if

return civDate
```

5. Menus

There are two menu types in the MILDB application. The first, such as menu m_MDIframe, is intended for a permanent display on user's interface.They continuously provide the menu's functionality to the user. The second type, such as menu

m_adminObject, is instantiated and provides its list of commands in the form of a popup menu only when needed.

Each of these menu types have an ancestor version, which serves MILDB users who have access to data from only a single Unit, and a descendant version, which provides additional functionality to global users who can access records from several Units. The descendant menu names carry an extension "_tc", for "troop command".

a. Menu m_MDI

This menu is displayed permanently along with the MDI (Multiple Display Interface) frame, and provides lists of commands in three major groups:

- File (includes commands, such as Save, Print, Close Sheet, Exit, etc.).
- Folder (opens the initial window for distinct MILDB operations, such as editing student administrative data, physical training-related data, and dormitory room assignment).
- Help (opens user help, and the 'About' window).

Figure 7 on the following page shows the structure of menu m_MDI, and its descendant menu m_MDI_tc.

The following code shows an example of scripts that drive both menus:

```
//***MENU m_MDIframe

MENU m_file.m_save

window activeSheet              //active window
commandButton currButton        //command button
int I                           //step counter

activeSheet = w_frame.GetActiveSheet()

if isValid(activeSheet) then
   for i = 1 to upperBound(activeSheet.control[])
      if activeSheet.control[i].typeOf() = commandButton! then
         currButton = activeSheet.control[i]
         if string(currButton.classname()) = "cb_save" then
            triggerEvent(currButton, clicked!)
            return
         end if
      end if
   next
end if


MENU m_file.m_close

window activeSheet              //active window

activeSheet = w_frame.GetActiveSheet()

if isValid(activeSheet) then
   close(activeSheet)
   if isValid(w_frame.GetActiveSheet()) = false then
      w_frame.MDI_1.resize(0,0)
   end if
else
   //openSheet(     )
   w_frame.MDI_1.resize(0,0)
end if
```

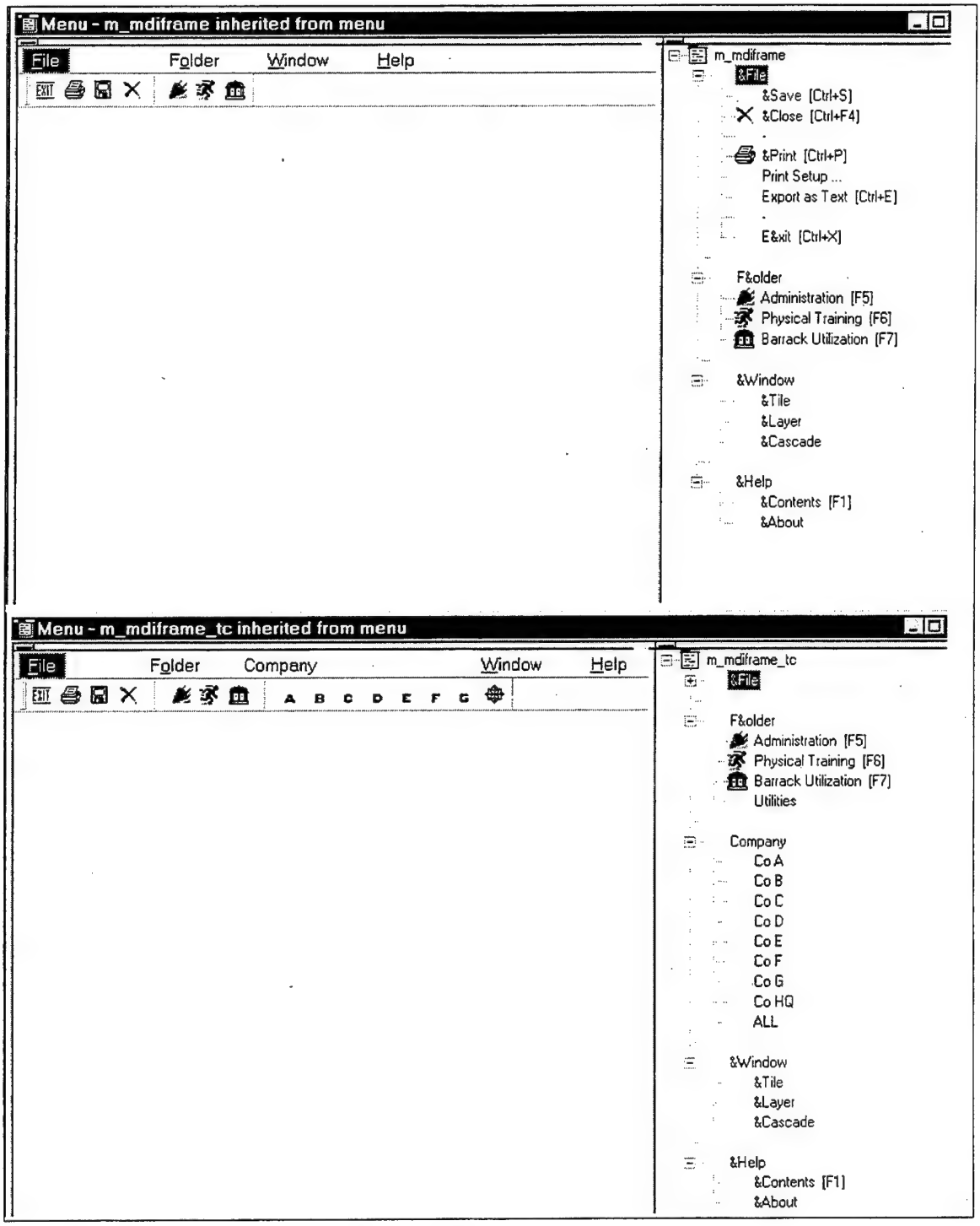


Figure 7. Menu m_MDI and m_MDI_tc

MENU m_file.m_print

```
w_base currWind              //active window

currWind = w_frame.GetActiveSheet()

if isValid( currWind) then
   currWind.printReport()
   return
end if


messageBox("Print", "No report to print.")
```

MENU m_file.m_exportastext

```
window activeSheet               //active window
commandButton currButton         //button
int i                            //local counter

activeSheet = w_frame.GetActiveSheet()

if isValid(activeSheet) then
   for i = 1 to upperBound(activeSheet.control[])
      if activeSheet.control[i].typeOf() = commandButton! then
         currButton = activeSheet.control[i]
         if string(currButton.classname()) = "cb_export" then
            triggerEvent(currButton, clicked!)
            return
         end if
      end if
   next
end if
```

MENU m_file.m_exit

```
close(parentWindow)
```

MENU m_folder.m_admin

```
setPointer(hourGlass!)
OpenSheet(w_admin, w_frame, 1, layered!)
```

MENU m_folder.m_physicaltraining

```
setPointer(hourGlass!)
openSheet(w_apft_wc, w_frame, 1, layered!)
```

```
MENU m_folder.m_barrackbunkassignment

setPointer(hourGlass!)
openSheet(w_hotel_acc, w_frame, 1, layered!)
```

b. Menu m_adminObject

Menu m_adminObject is not permanently displayed. It contains commands for opening windows that serve to accomplish a specific task, such as to create a new student record, to record a pregnancy counseling session, or to display a name roster. They appear as popup menus when needed. Names of menus and submenus were chosen to also indicate their function. Figures 8 and 9 show the structure of menu m_adminObject, and its descendant menu m_adminObject_tc.

The following code shows an example of script that drives the menus:

```
//***MENU ADMINOBJECT

MENU m_inprocessing.m_pt1

int success    //open window success/fail

success = openSheet(w_admin, w_frame, 1, layered!)

if success = 1 then
   w_admin.setW_admin( this.text)
   if fetch_ssn <> "" then
      w_admin.fetchData()
   end if
end if
```

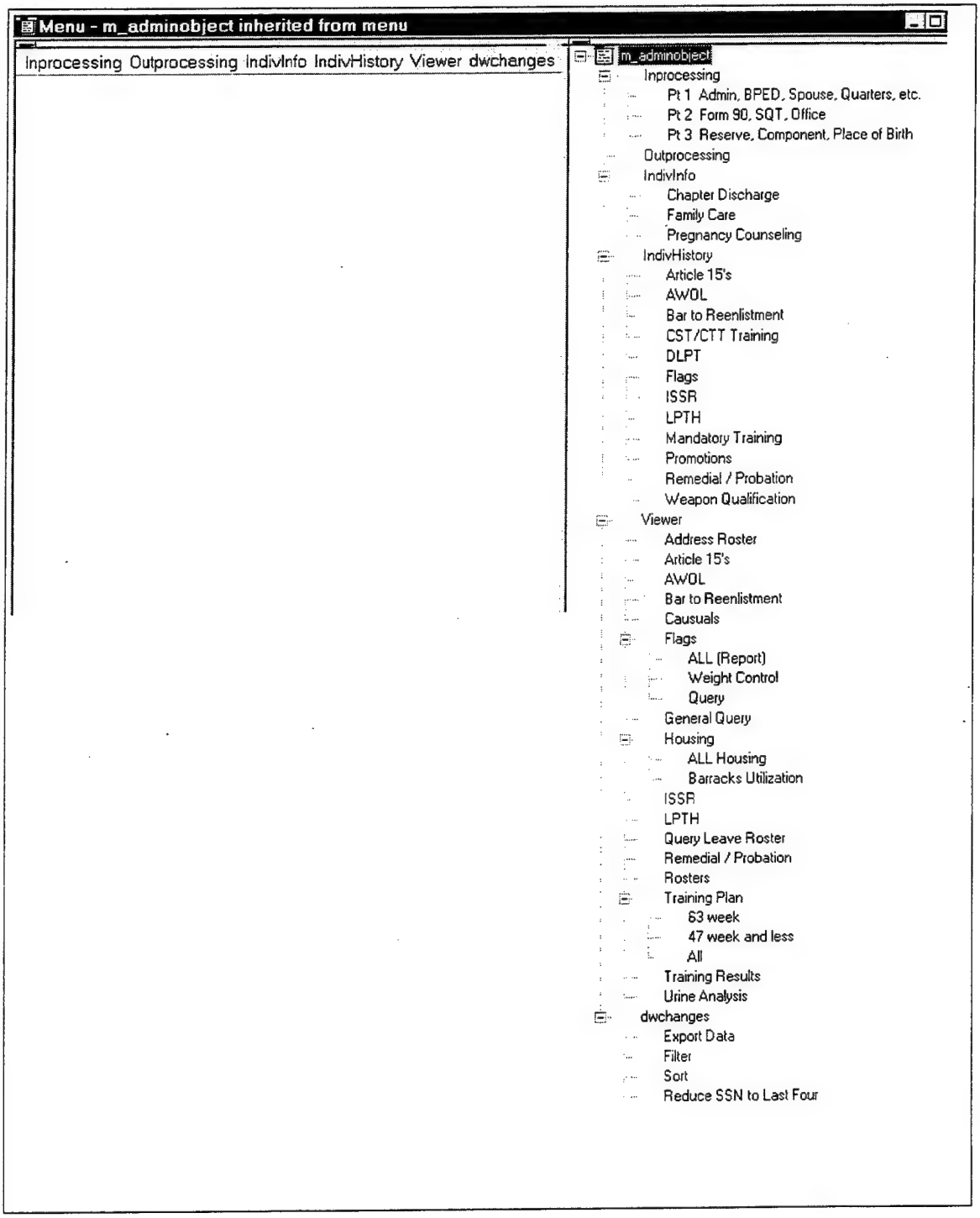


Figure 8. Menu m_adminObject

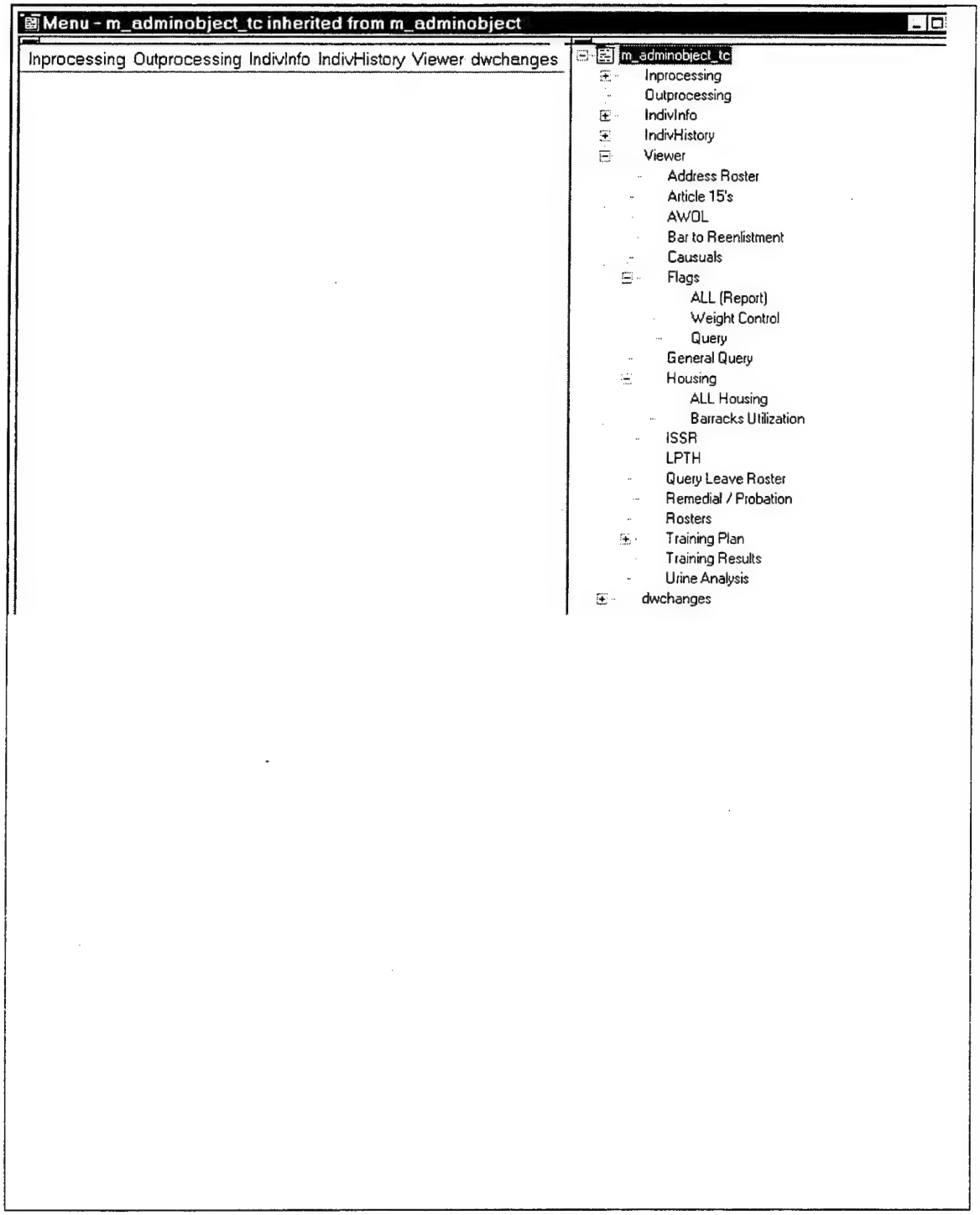


Figure 9. Menu m_adminObject_tc

MENU m_inprocessing.m_pt2

```
int success     //open window success/fail

success = openSheet(w_admin, w_frame, 1, layered!)

if success = 1 then
   w_admin.setW_admin( this.text)
   if fetch_ssn <> "" then
      w_admin.fetchData()
   end if
end if
```

MENU m_inprocessing.m_pt3

```
int success     //open window success/fail

success = openSheet(w_admin, w_frame, 1, layered!)

if success = 1 then
   w_admin.setW_admin( this.text)
   if fetch_ssn <> "" then
      w_admin.fetchData()
   end if
end if
```

6. User Objects

Two major user objects are used throughout the MILDB application: Locator (named o_locate) which displays list of personnel and triggers retrieval of data pertinent to a given person, and Admin user object (named o_adminObject) which serves as visual interface for displaying popup menus that are related to a specific administrative task.

a. Locator o_locate

This object retrieves and displays a list of platoons and names of all personnel assigned to a single Unit. Platoon names are displayed on the left side of the Locator in dataWindow dw_plt, one platoon name per row. Names of the personnel are displayed on the right side of the Locator, in dataWindow dw_loco, one name per row. When user clicks on name of a platoon, list of personnel assigned to that platoon appears in dw_loco. When checkBox 'Show All', which appears at the bottom of the Locator, is selected, all names of personnel in the Unit appear in alphabetical order in dw_loco. When user clicks a person's name in dw_loco, that person's SSN is submitted to the currently active window for further processing. Figure 10 shows the Locator.

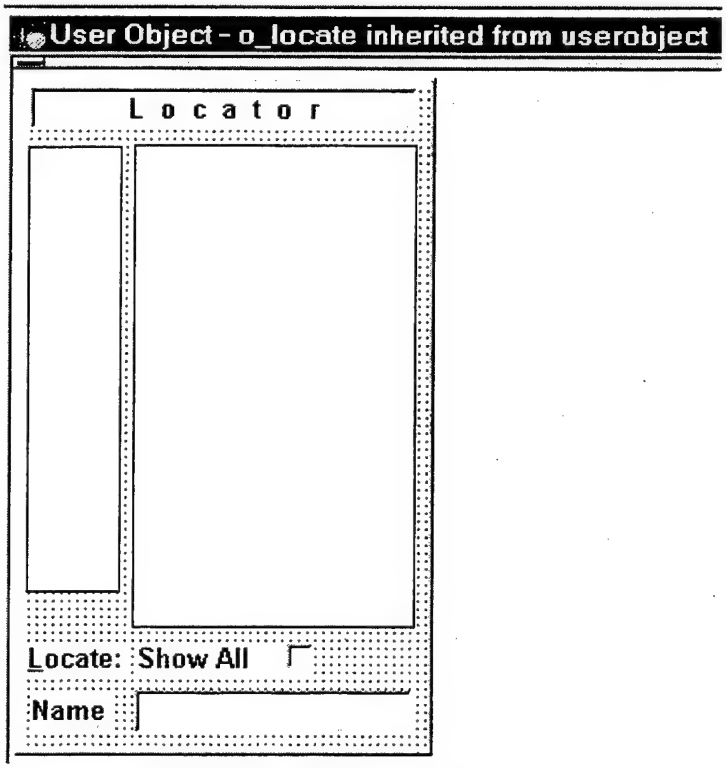


Figure 10. Locator o_locate

The following code shows script of events of the object o_locate, and of the controls contained in it:

```
//***USER OBJECT O_LOCATE
//IS USED ENYWHERE WHERE LIST OF STUDENTS IN UNIT IS NEEDED

//INSTANCE VARIABLES
m_adminObject  newMenu     //instance of menu with supporting functions
long           oldRow = 0  //saves previous row indicator


//OBJECT EVENTS

//EVENT CONSTRUCTOR
string comp

//instantiate menu
newMenu = create m_adminObject

comp = upper(left(company, 1))
if comp = "%" or comp = "" then
   comp = "ALL"
end if
st_2.text = "Company  " + comp

cbx_1.checked = false

//set transaction object for dataWindows
dw_plt.SetTransObject(sqlca)
dw_loco.setTransObject(sqlca)

//retrieve students
dw_plt.retrieve(company)


//EVENT DESTRUCTOR
dw_plt.reset()
dw_loco.reset()

destroy newMenu


//OBJECT FUNCTIONS

//PARAMETERS:  string   ssn
//             string   filterStrng - dataWindow filter definition
//RETURN:      none
//PURPOSE:     Finds record for given SSN in dw_loco,
//             highlights proper platoon in dw_plt, and name in dw_loco
```

```
showLine( string SSN, readonly string filterStrng)

//Local variables
string plt        //platoon
string Bucket     //temp

long pltRow       //row number in dw_plt
long locoRow      //row number in dw_plt


//lock display
dw_plt.setRedraw( false)
dw_loco.setRedraw( false)

//unselect any platoon row
dw_plt.selectRow( 0, false)
dw_loco.selectRow( 0, false)

//reset filter for names display
dw_loco.setFilter( "")
dw_loco.Filter()

//find platoon for ssn
Bucket = "ssn = ~"" + ssn + "~""
locoRow = dw_loco.find( Bucket, 1, 10000)

if locoRow > 0 then

   plt = dw_loco.getItemString( locoRow, "plt")
   Bucket = "plt = ~"" + plt + "~""
   pltRow = dw_plt.find( Bucket, 1, 1000)

   //display records for platoon
   Bucket = "plt = ~"" + plt + "~""
   if (not isNull( filterStrng)) and ( filterStrng <> "") then
      Bucket += " and " + filterStrng
   end if
   dw_loco.setFilter( Bucket)
   dw_loco.filter()

   //find ssn in filtered dw_loco
   Bucket = "ssn = ~"" + ssn + "~""
   locoRow = dw_loco.find( Bucket, 1, 10000)

   //highlight rows for appropriate platoon and ssn
   dw_plt.selectRow(  pltRow, true)
   dw_loco.selectRow( locoRow, true)

else
   //ssn not found => move all rows to filter buffer
   dw_loco.RowsMove(1, dw_loco.rowCount(), PRIMARY!, dw_loco, 10000,
FILTER!)
end if
```

```
//unlock display
dw_plt.setRedraw( true)
dw_loco.setRedraw( true)

return



//PARAMETERS:  long  currRow - current row indicator
//RETURN:      none
//PURPOSE:     Submits SSN to a window for further processing
submitToForm( readonly long currRow)

//Local variables
string currWindowNm  //name of current window

if currRow > 0 then
   fetch_ssn = dw_loco.getItemString( currRow, 2)
   sle_ssn.text = fetch_ssn
else
   return
end if

currWindowNm = w_frame.getActiveSheet().classname()

choose case currWindowNm

   case "w_mildata"
      w_mildata.fetchData()

   case "w_history"
      w_history.fetchData()

   case "w_admin"
      w_admin.fetchData()

   case "w_weightcontrol"
      w_weightcontrol.fetchData()

   case "w_apft"
      w_apft.fetchData()

   case "w_profile"
      w_profile.fetchData()

   case "w_wthist"
      w_wthist.fetchData()

   case "w_apfthist"
      w_apfthist.fetchData()

   case else
      //do nothing
end choose
```

```
this.borderStyle = styleBox!


//EVENTS OF CONTROLS

//CHECK BOX cbx_1
//EVENT CLICKED
//When the checkBox state 'Checked' => show ALL personnel in dw_loco
//otherwise => show personnel by platoon
dw_loco.setRedraw(false)

if this.checked = false then
   dw_loco.setFilter("plt = ~"-?~" ")
   dw_loco.Filter()
else
   dw_plt.selectrow(0, false)

   if dw_loco.filteredCount() = 0 then
      dw_loco.retrieve(company)
   end if

   dw_loco.setFilter("isNumber(ssn)")
   dw_loco.Filter()
   dw_loco.selectrow(0, false)
   dw_loco.selectrow(1, true)
   dw_loco.setFocus()

end if

dw_loco.setRedraw(true)


//DATAWINDOW dw_loco
//CONTAINS NAME AND SSN OF STUDENTS
//EVENT CLICKED
//When the checkBox state 'Checked' => show ALL personnel in dw_loco
//otherwise => show personnel by platoon
dw_loco.setRedraw(false)

if this.checked = false then
   dw_loco.setFilter("plt = ~"-?~" ")
   dw_loco.Filter()
else
   dw_plt.selectrow(0, false)

   if dw_loco.filteredCount() = 0 then
      dw_loco.retrieve(company)
   end if

   dw_loco.setFilter("isNumber(ssn)")
   dw_loco.Filter()
```

```
   dw_loco.selectrow(0, false)
   dw_loco.selectrow(1, true)
   dw_loco.setFocus()

end if

dw_loco.setRedraw(true)


//EVENT RIGHTBUTTOTDOWN
//When the checkBox state 'Checked' => show ALL personnel in dw_loco
//otherwise => show personnel by platoon
dw_loco.setRedraw(false)

if this.checked = false then
   dw_loco.setFilter("plt = ~"-?~" ")
   dw_loco.Filter()
else
   dw_plt.selectrow(0, false)

   if dw_loco.filteredCount() = 0 then
      dw_loco.retrieve(company)
   end if

   dw_loco.setFilter("isNumber(ssn)")
   dw_loco.Filter()
   dw_loco.selectrow(0, false)
   dw_loco.selectrow(1, true)
   dw_loco.setFocus()

end if

dw_loco.setRedraw(true)


//EVENT KEYDOWN
//Action to take when user presses certain keys

long currRow

//keyEnter
if keyDown( keyEnter!) then
   //start processing
   submitToForm( currRow)
//keyControl
elseif keyDown( keyControl!) then
   currRow = getSelectedRow( 0)
   //keyDownArrow
   if keyDown( keyDownArrow!) then
      //select the row above
      this.selectRow( 0, false)
      this.selectRow( currRow + 1, true)
```

```
      submitToForm( currRow +1 )
   end if
   //keyUpArrow
   if keyDown( keyUpArrow!) then
      //select the row below
      this.selectRow( 0, false)
      this.selectRow( currRow - 1, true)
      submitToForm( currRow - 1 )
   end if

end if



//EVENT MOUSEMOVE
//highlight rows as the user moves the mouse over rows

string   rowStrng    //row identifier
long     pos         //position of character

rowStrng = this.getObjectAtPointer()
pos = Pos( rowStrng, "~t", 1)
pos += 1
rowStrng = mid( rowStrng, pos)
pos = long( rowStrng)

if pos > 0 then
   if pos <> oldRow then
      This.SelectRow(0, FALSE)
      this.selectRow( pos, true)
      oldRow = pos
   end if
end if



//DATAWINDOW dw_plt
//CONTAINS PLATOON NAMES
//EVENT CLICKED
string   platoon        //name of platoon
string   currWindowNm   //name of current window
string   sex            //gender
string dwFilter         //filter definition for dataWindow

//hightlight current row
if row > 0 then
   this.selectrow(0, false)
   this.selectrow( row, true)
end if

if row > 0 then

   cbx_1.checked = false
```

```
    platoon = trim( this.getitemstring( row, 1))

    if isNull(platoon) then
       platoon = "%"
    end if

    currWindowNm = lower( w_frame.getActiveSheet().classname())

    //show only Males, or Females, or ALL
    choose case currWindowNm

       case "w_weightcontrol"
          //get only Male or Female
          if w_weightControl.tab_wc.selectedTab = 1 then
             //sex = "M"
             dwFilter = "plt = '" + platoon + "' and sex = ~"M~" "
          else
             //sex = "F"
             dwFilter = "plt = '" + platoon + "' and sex = ~"F~" "
          end if

       case else
          dwFilter = "plt = '" + platoon + "'"

    end choose

    dw_loco.setRedraw(false)

    if (dw_loco.rowCount() > 0) OR (dw_loco.filteredCount() > 0) then
       //reset the filter, before new filter will be applied
       //this will ALL personnel back into dw_loco
       dw_loco.setFilter("")
       dw_loco.Filter()
    else
       //if personnel not retrieved yet, do it
       dw_loco.retrieve(company)
    end if

    //set and apply new filter
    dw_loco.setFilter(dwFilter)
    dw_loco.Filter()

    dw_loco.selectRow(1, true)
    dw_loco.setRedraw(true)
    dw_loco.setFocus()

end if
```

b. User Object o_adminObject

User object o_admin appears permanently on windows that provide the interface for fulfilling some administrative tasks, such as entering new service member into the database, updating person's administrative data, outprocessing a student, deactivating his/her record, etc. The purpose of this object is to display the choice of several the major administrative task options available to the user, and to display a popup menu offering further options once a specific task is selected (clicked). Afterbeing selected, the background color of a label that displays an administrative task changes, and remains highlighted even when a popup menu is dismissed and the user proceeds to work with displayed data. This provides the user with persisting visual clue about the nature of the operation that he/she is performing. Replacing this user object with a simple menu would not provide such visual clue. Figure 11 shows the design of object o_adminObject and menus that popup when its label, indicating an administrative task, is clicked.

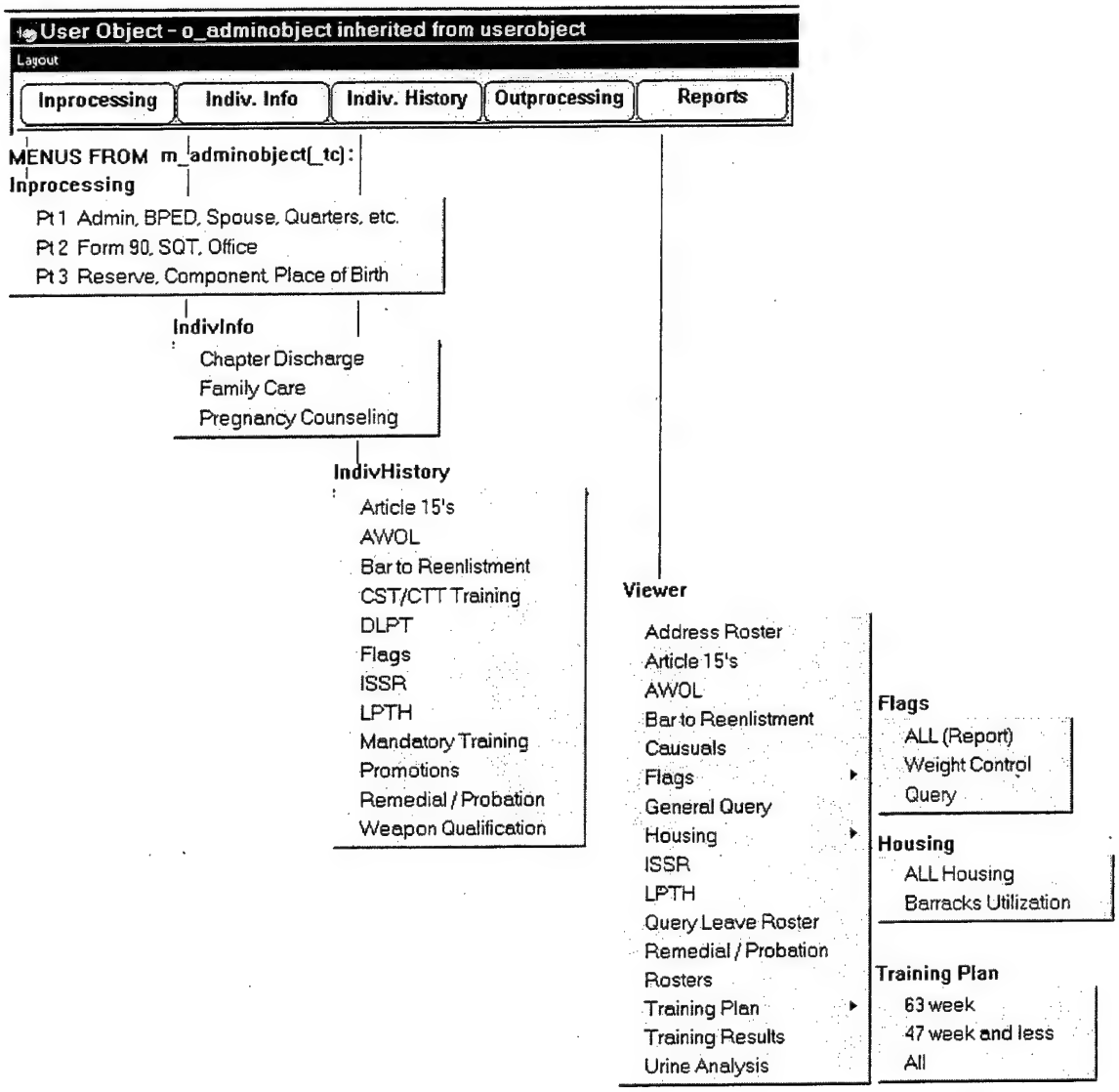


Figure 11. User Object o_adminObject

c. User Object u_storedProc

This is a non-visual user object that is used for database transactions, and also as an interface for executing stored procedures. How such object can be created was described in Section C. Figure 12 shows the external functions and subroutines declared in u_storedProc.

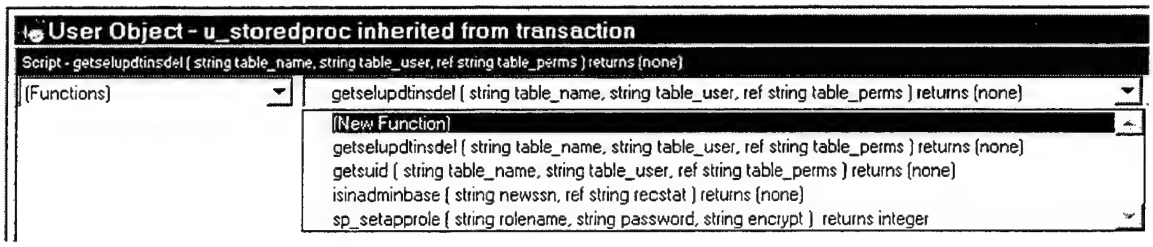


Figure 12. User Object u_storedProc

7. Windows

All windows in the MILDB application are hosted by a window of type MDI. Based on user's action, other windows are opened as sheets within the MDI frame. To demonstrate the process of designing and scripting MILDB windows, the MDI frame and one major window, which implement inheritance, is documented in this thesis.

a. MDI Frame w_frame

This window hosts other windows, called sheets, of MILDB application. Associated with this window is menu m_MDIframe(_tc) described earlier in this Section. Figure 13 shows the design of window w_frame.

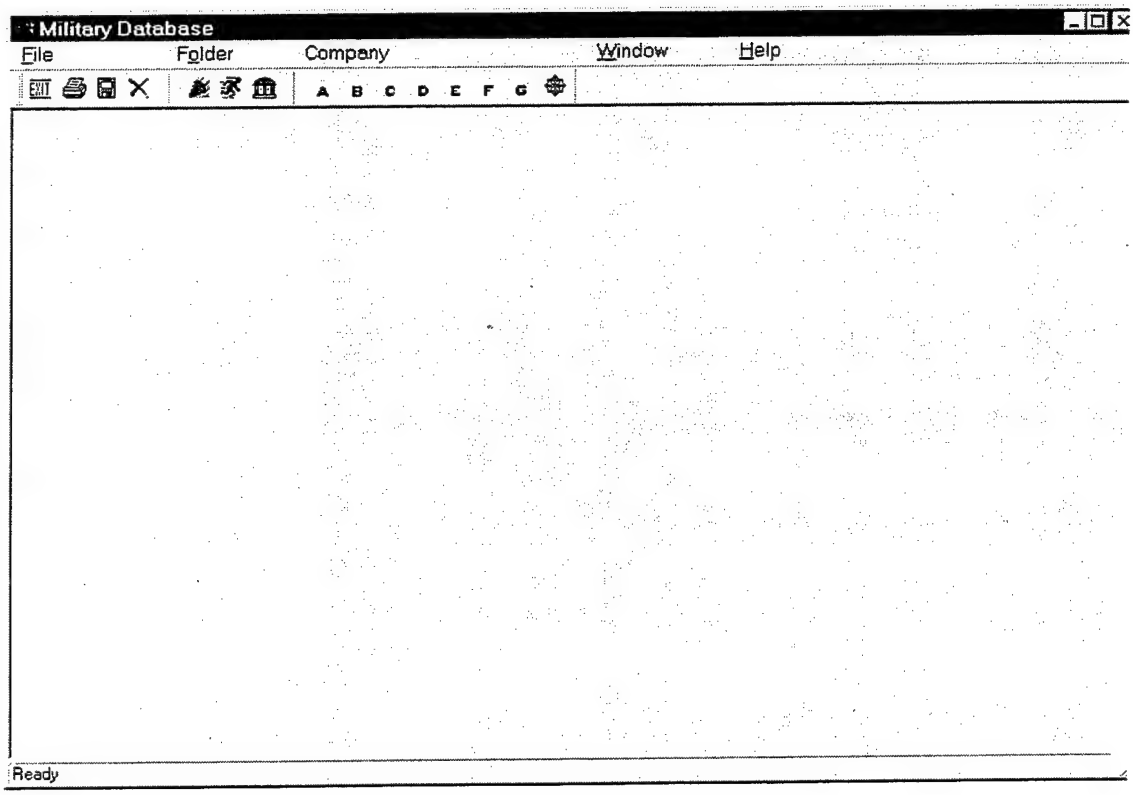


Figure 13. Window w_frame

The following code shows script of the opening event of w_frame:

```
//***WINDOW W_FRAME
//WINDOW OPEN EVENT

//Local variables
string errStrng            //error string
string Bucket              //temp

long rowCnt                //row count

dataStore ds_tempStore     //non-visual dataStore

//instantiate dataStore
ds_tempStore = CREATE dataStore

//sql syntax for dataStore
//select Unit(s) and Platoon(s) that USER can see
Bucket = "select unit from mil.v_cansee"

Bucket = sqlca.SyntaxFromSQL( Bucket, "", errStrng)

//create dataStore and retrieve data
ds_tempStore.create( Bucket, errStrng)
ds_tempStore.setTransObject( sqlca)
rowCnt = ds_tempStore.retrieve()

choose case rowCnt
   case 0
      messageBox( "MILDB OPEN", "Can't see any information.")
      //quit application
      HALT

   case is > 3
      //show troop command menu
      //intialize global variable to "A"
      company = "A"
      this.changeMenu(m_mdiframe_tc)
      Bucket = "ALL COMPANY DATABASE"

   case else
      //set global variable
      company = ds_tempStore.getItemString( 1, "unit")
      //set Unit-level menu
      this.changeMenu(m_mdiframe)
      Bucket = "COMPANY " + company + " DATABASE"

end choose

//set the window title
this.title = Bucket
w_splash.bringToTop = true
```

b. Window w_base

This window is an ancestor of numerous other windows found in MILDB. It contains a single object, user object o_locate, described earlier in this section. In this window are declared several window-level functions whose scripts are defined, or extended, in descendant windows. Figure 14 shows the design of window w_base.

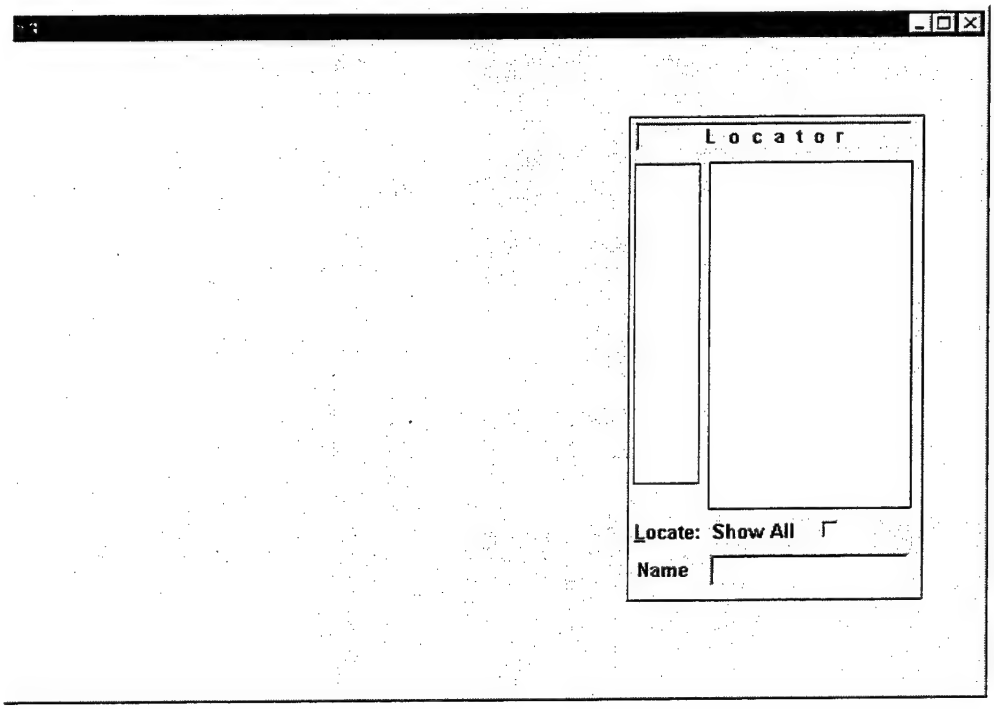


Figure 14. Window w_base

The following code shows scripts of w_base events and functions:

```
//***WINDOW W_BASE

//DECLARE INSTANCE VARIABLES
datawindow dwList[]      //list of dataWindow controls
int closeCode            //0=can, 1=cannot close window
boolean saveEnabled      //stores the state of menuItem 'Save'


//WINDOW EVENTS

//WINDOW OPEN EVENT
int xx, yy    //local counters

//call window function (get number of data windows)
yy = getDwList( this, dwList)

if yy > 0 then

   //hide all dataWindows
   for xx = 1 to yy
      dwList[xx].hide()
   next

   //create menu for dataWindow utilities
   if isValid( dwUtilMenu) then
      //do nothing
   else
      dwUtilMenu = CREATE m_dwchanges
   end if
end if


//WINDOW ACTIVATE EVENT
//restore the last state of 'Save' menu in MDI frame
w_frame.menuID.item[1].item[1].enabled = saveEnabled


//WINDOW DEACTIVATE EVENT
//save the last state of 'Save' menu in MDI frame
saveEnabled = w_frame.menuID.item[1].item[1].enabled


//KEY DOW EVENT
//set focus to locator
if keyDown( keyControl!) then
   uo_1.dw_loco.setFocus()
end if
```

```
//MOUSE MOVE EVENT
if uo_1.borderStyle <> styleBox! then
   uo_1.borderStyle = styleBox!
   uo_1.bringToTop = false

   if isValid( getFocus()) then
      if getFocus().className() = "dw_loco" then
         dwList[upperBound(dwList)].setFocus()
      end if
   end if
end if


//CLOSE QUERY EVENT

//Initialize instance variable
closeCode = 0      //allow closing

//Local variables
int msgCode        //return code from messageBox
int dwCnt          //row count
int i              //step counter

long modifiedCnt   //number of modified rows
modifiedCnt = 0

//check if 'Save' enabled in MDI menu
if w_frame.menuID.item[1].item[1].enabled = false then
   return 0
end if

//check for data change
dwCnt = upperBound(dwList[])
for i = 1 to dwCnt
   dwList[i].acceptText()
   modifiedCnt += dwList[i].modifiedCount()
next

if modifiedCnt > 0 then

   msgCode = messageBox (this.title, "Do you want to save changes" &
                         + " to current record?", &
                         Question!, YesNoCancel!)

   CHOOSE CASE msgCode
   CASE 1
      closeCode = 0 //close after saving
      saveData()
   CASE 2
      closeCode = 0 //close without saving
   CASE ELSE
      closeCode = 1 //don't close
   END CHOOSE
end if
```

```
//allows / aborts closure of window
return closeCode


//WINDOW FUNCTIONS

//PARAMETERS:  none
//RETURN:      (none)
//PURPOSE:     Enable/disable SAVE menu based on permissions
setSaveMenu()

//Local variables
string updtTable  //table name
string permList   //permission string SUID
                  //S - Save, U - Update, I - insert, D - delete
string Bucket     //temp

int xx            //step counter

if SQLCA.Database = "mildb" then
   //update possible
   //First menu item in 1st submenu is 'Save'
   w_frame.menuID.item[1].item[1].enable()
   return
end if

//get permissions from all dataWindows except dw_1
permList = ""

for xx = 1 to upperBound( dwList[])
   if dwList[ xx].dataObject <> "" then
      updtTable = dwList[ xx].Object.dataWindow.Table.updateTable

      if updtTable <> "" then
         updtTable = mid( updtTable, pos( updtTable, ".") + 1)
         Bucket = "????"
         sqlca.getSUID( updtTable, sqlca.userID, Bucket)
         permList += Bucket
      end if
   end if
next

//now set the menu
if pos( permList, "U") > 0 then
   //update possible
   //First menu item in 1st submenu is 'Save'
   w_frame.menuID.item[1].item[1].enable()

else
   //update restricted
   //First menu item in 1st submenu is 'Save'
   w_frame.menuID.item[1].item[1].disable()
end if
```

```
return


//PARAMETERS:   none
//RETURN:       (none)
//PURPOSE:      Retrieve records
fetchData()
//do nothing
//declared here, but will be extended in descendent windows


//PARAMETERS:   none
//RETURN:       (none)
//PURPOSE:      Print report
printReport()
//do nothing
//declared here, but will be extended in descendent windows


//PARAMETERS:   none
//RETURN:       (none)
//PURPOSE:      Update dataWindows
savedata()
//do nothing
//declared here, but will be extended in descendent windows


//button cb_print
//event clicked()
parent.printReport()
```

c. Window w_base_admin

This window is a descendant of w_base. It adds user object o_adminObject to objects already contained in the ancestor window. The window's Open event is extended to include some initial setup of its user objects. This window is an immediate ancestor of windows handling administrative data processing of MILDB. Figure 15 shows the design of w_base_admin.

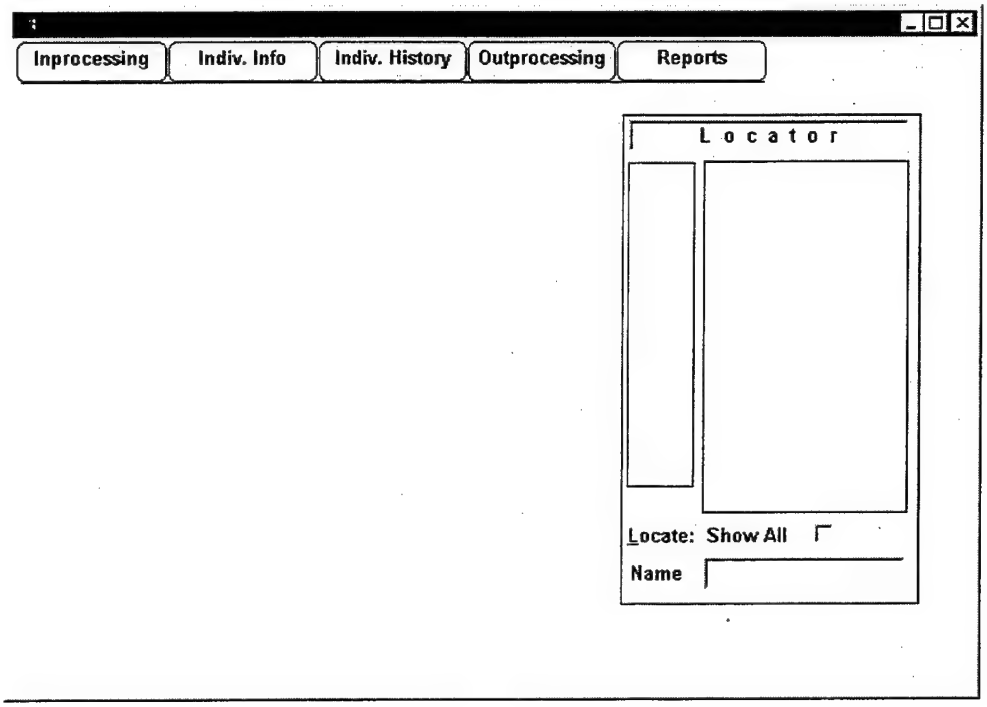


Figure 3. Window w_base_admin

The following code shows scripts contained in w_admin_base:

```
//WINDOW W_BASE_ADMIN
//INHERITED FROM W_BASE


//WINDOW EVENTS

//WINDOW OPEN EVENT
//EXTENDS ANCESTOR SCRIPT

//LOCAL VARIABLES
long lghtGray           //RGB
string lghtGrayStrng    //RGB string
```

```
//count instances of admin windows
adminCnt ++

//initialize adminMenu shared variable
if not isValid( adminMenu) then
   adminMenu = create m_adminObject_tc
end if

lghtGray = RGB(192, 192, 192)
lghtGrayStrng = string( lghtGray)

//set bkgr color of tabs in action menu
uo_adminmenu.st_inProcessing.backColor = lghtGray  //light gray
uo_adminmenu.st_outProcessing.backColor = lghtGray //light gray
uo_adminmenu.st_indivInfo.backColor = lghtGray     //light gray
uo_adminmenu.st_indivHistory.backColor = lghtGray  //light gray
uo_adminmenu.st_viewer.backColor = lghtGray        //light gray

uo_adminmenu.st_inProcessing.tag = lghtGrayStrng   //light gray
uo_adminmenu.st_outProcessing.tag = lghtGrayStrng  //light gray
uo_adminmenu.st_indivInfo.tag = lghtGrayStrng      //light gray
uo_adminmenu.st_indivHistory.tag = lghtGrayStrng   //light gray
uo_adminmenu.st_viewer.tag = lghtGrayStrng         //light gray


//WINDOW RESIZE EVENT
//set position of Locator, based on window/screen size
uo_1.X = this.width - uo_1.width - 50
```

d. Window w_admin

This window is a descendant of window w_base_admin. In addition to objects it has inherited from its ancestors, it also includes dataWindow controls for data retrieval and manipulation, and a button for creating new records. Several window-level functions, declared in its ancestor windows, are defined here, or are just extended. New window-level functions, performing the initial setup of the window, are added. Figure 16 shows the design of the window, and an example of data display (all data are fictional).

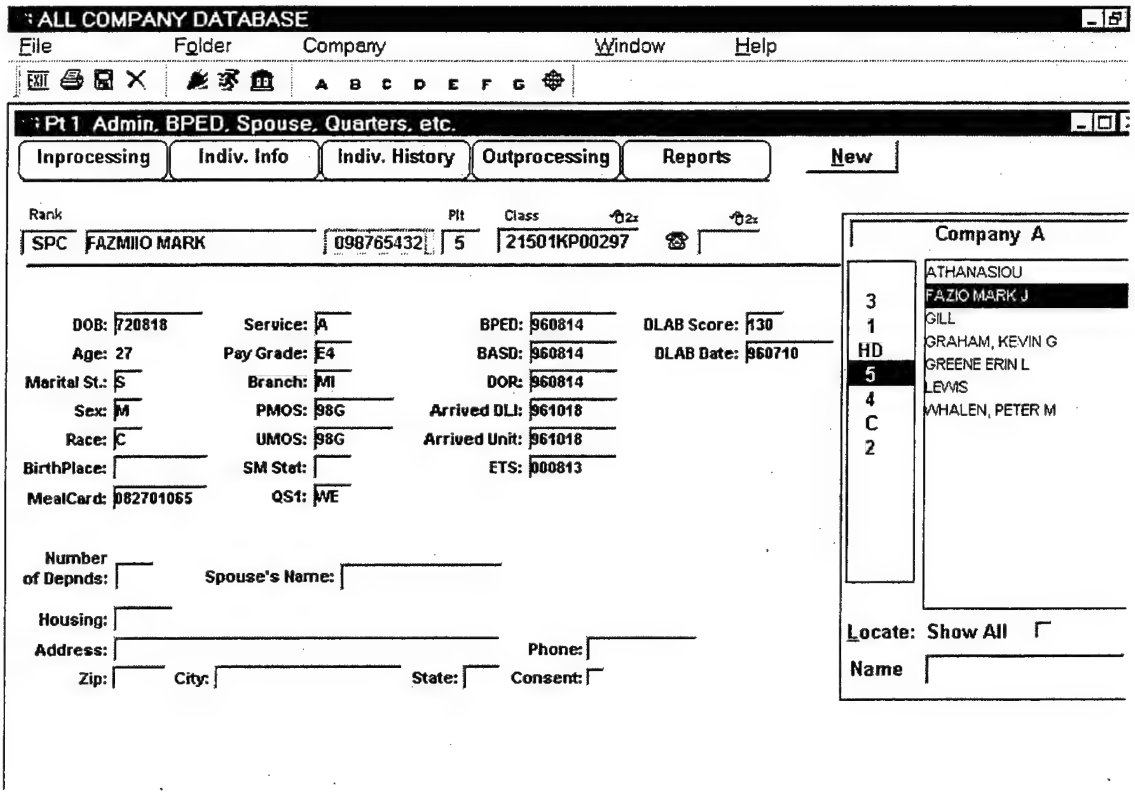


Figure 16. Window w_admin

The following code shows scripts of window w-admin events, and window-level functions.

```
//***WINDOW W_ADMIN
//DESCENDANT OF W_BASE_ADMIN

//DECLARE ADDITIONAL INSTANCE VARIABLES
//none


//WINDOW EVENTS

//WINDOW OPEN EVENT
//EXTENDS ANCESTOR SCRIPT
```

```
//Local variables
string   Bucket      //temp

//set window title
this.title = " Inprocessing "

//find if user can inser new soldier
Bucket = space(4)
sqlca.getSUID( "admin", sqlca.userID, Bucket)
if pos( Bucket, "I") > 0 then
   cb_add.show()
else
   cb_add.hide()
end if

//set labels in 'tab' user object
uo_adminmenu.st_inProcessing.backColor = RGB(255, 255, 255)   //white
uo_adminmenu.st_inProcessing.tag = string(RGB(255, 255, 255)) //white


//WINDOW FUNCTIONS

//EXTENDS ENCESTOR FUNCTION
fetchData()

int   xx    //STEP COUNTER

//check for modification of previously retrieved data
//triggerEvent( closeQuery!)

if closeCode = 1 then
   //user doesn't want to move to next record
   return
end if

xx = setW_admin( this.title)
if xx <> 0 then
   //no admin form selected yet
   return
end if

setRedraw( false)

//reset dataWindow
for xx = 1 to upperBound( dwList)
   dwList[xx].reset()
next

//retrieve data
if fetch_ssn <> "" then
   dw_1.retrieve( fetch_ssn)
   dw_2.retrieve( fetch_ssn)
   dw_3.retrieve( fetch_ssn)
   dw_4.retrieve( fetch_ssn)
```

```
else
   return
end if

if dw_1.rowCount() = 1 then
   primeNewRows( fetch_ssn)
else
   messageBox( "ADMIN DATA", "No recors retrieved.")
end if

dw_1.show()
dw_2.show()
dw_3.show()
dw_4.show()
dw_5.hide()
dw_6.hide()

if dw_2.dataObject = "d_form90" then
   dw_6.retrieve( fetch_ssn)
   dw_6.show()
end if

setRedraw( true)

return


//EXTENDS ANCESTOR SCRIPT
printReport()

setpointer(hourglass!)

if dw_1.rowCount() < 1 then
   messageBox("PRINT REPORT", "Nothing to print.")
   return
end if

string   Bucket                    //temp
string   equals, dashes, blnk //text containers

int      job                       //print job
int      t1, t2, t2a, t3, t3b, t4, t5, t6 //print indents

t1 = 300
t2a = 1200
t2 = 2400
t3 = 5000
t3b = 4500
t4 = 6500
t5 = 6700
//t6 = 6000
```

```
if (dw_2.dataObject = "d_admin") OR (dw_2.dataObject = "d_adminnew")
then

   job = printopen()

   printDefineFont( job, 1, "Arial", -10, 400, default!,&
                    anyfont!, false, false)
   printDefineFont( job, 2, "Arial", -10, 700, default!,&
                    anyfont!, false, false)

   dashes =
"_____________________________________________________________________
_____________________________________"
   equals =
"=====================================================================
====================================="
   blnk = "_________"

   //***header
   printSetFont( job, 1)
   print( job, 2500, "*** FOR OFFICIAL USE ONLY - PRIVACY ACT DATA
***")
   print( job, "")
   print( job, "")
   print(job, 3200, "PERSONAL DATA REPORT")
   print(job, "")
   print(job, "")
   print(job, "")
   print(job, "")

   print( job, 0, equals)
   printSetFont( job, 2)

   if (dw_2.dataObject = "d_admin") then
      Bucket =  dw_1.getitemstring(1, "rank")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, t1, "Rank: " + Bucket, t2a )
      Bucket = dw_1.getitemstring(1, "n_student")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, "Name: " + Bucket, t3b )
      Bucket = dw_1.getitemstring(1, "ssnadmin")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, "SSN: " + Bucket, 2*t2 + 1000)
      Bucket = dw_1.getitemstring(1, "plt")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, "PLT: " + Bucket, t4)
      Bucket = dw_1.getitemstring(1, "class")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, "Class: " + Bucket )
      print(job, "")
   else
      Bucket =  dw_2.getitemstring(1, "rank")
      if isNull(Bucket) then; Bucket = ""; end if
      print(job, t1, "Rank: " + Bucket, t2a )
```

```
        Bucket = dw_2.getitemstring(1, "n_student")
        if isNull(Bucket) then; Bucket = ""; end if
        print(job, "Name: " + Bucket, t3b )
        Bucket = dw_2.getitemstring(1, "ssntarget")
        if isNull(Bucket) then; Bucket = ""; end if
        print(job, "SSN: " + Bucket, 2*t2 + 1000)
        Bucket = dw_2.getitemstring(1, "plt")
        if isNull(Bucket) then; Bucket = ""; end if
        print(job, "PLT: " + Bucket, t4)
        Bucket = dw_2.getitemstring(1, "class")
        if isNull(Bucket) then; Bucket = ""; end if
        print(job, "Class: " + Bucket)
        print(job, "")
        print(job, "")
    end if

    printSetFont( job, 1)
    print( job, 0, equals)
    if (dw_2.dataObject = "d_adminnew") then
       printSetFont( job, 2)
    end if
    print(job, "")
    print(job, "")

//LINE1
    Bucket = string(date(dw_2.getitemDateTime(1, "dob")))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, t1, "DOB: " + Bucket, t2 )
    Bucket = dw_2.getitemstring(1, "service")
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "Service: " + Bucket, t3b )
    Bucket = string(date(dw_2.getitemDateTime(1, "bped")))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "BPED: " + Bucket, t4 )
    Bucket = string(date(dw_2.getitemDateTime(1, "d_dlab")))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "DLAB Date: " + Bucket)
    print(job, "")

//LINE2
    Bucket = string(dw_2.getitemNumber(1, "age"))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, t1, "Age: " + Bucket, t2 )
    Bucket = dw_2.getitemstring(1, "pay_grd")
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "Pay Grade: " + Bucket, t3b )
    Bucket = string(date(dw_2.getitemDateTime(1, "basd")))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "BASD: " + Bucket, t4 )
    Bucket = string(dw_2.getitemNumber(1, "q_dlab"))
    if isNull(Bucket) then; Bucket = blnk; end if
    print(job, "DLAB Score: " + Bucket )
    print(job, "")
```

```
//LINE4
   Bucket = dw_2.getitemstring(1, "mar_stat")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, t1, "Marital St.: " + Bucket, t2 )
   Bucket = dw_2.getitemstring(1, "branch")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "Branch: " + Bucket, t3b )
   Bucket = string(date(dw_2.getitemDateTime(1, "d_rank")))
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "DOR: " + Bucket)
   print(job, "")

//LINE5
   Bucket = dw_2.getitemstring(1, "sex")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, t1, "Sex: " + Bucket, t2 )
   Bucket = dw_2.getitemstring(1, "pmos")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "PMOS: " + Bucket, t3b )
   Bucket = string(date(dw_2.getitemDateTime(1, "d_arrival")))
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "Arrived DLI: " + Bucket)
   print(job, "")

//LINE6
   Bucket  = dw_2.getitemstring(1, "race")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, t1, "Race: " + Bucket, t2 )
   Bucket = dw_2.getitemstring(1, "ult_mos")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "UMOS: " + Bucket, t3b )
   Bucket  = string(date(dw_2.getitemDateTime(1, "d_unit")))
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "Arrived Unit: " + Bucket )
   print(job, "")

//LINE7
   Bucket = dw_2.getitemstring(1, "mealcard")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, t1, "Mealcard: " + Bucket, t2 )
   Bucket = dw_2.getitemstring(1, "sm_status")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "SM Status: " + Bucket, t3b )
   Bucket = string(date(dw_2.getitemdateTime(1, "ets")))
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, "ETS: " + Bucket )
   print(job, "")

//LINE8
   Bucket = dw_2.getitemstring(1, "qs1")
   if isNull(Bucket) then; Bucket = blnk; end if
   print(job, t2, "QS1: " + Bucket)
   print(job, "")
   print(job, dashes)
```

```
   print(job, "")
   print(job, "")

//DEPN PART OF FORM

   Bucket = dw_3.getitemstring(1, "depn_n_spouse")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, t1, "Spouse's Name: " + Bucket, t3b )
   Bucket = dw_3.getitemString(1, "depn_numb_chil")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, "Number of Dependents: " + Bucket)
   print(job, "")
   print(job, dashes)
   print(job, "")
   print(job, "")


//LOCATION PART OF THE FORM
   Bucket = dw_4.getitemstring(1, "loc_addr")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, t1, "Address: " + Bucket, t3)
   Bucket = dw_4.getitemString(1, "loc_qtrs")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, "Quarters: " + Bucket)
   print(job, "")
   Bucket = dw_4.getitemString(1, "loc_city")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, t1, "City: " + Bucket, t2 )
   Bucket = dw_4.getitemString(1, "loc_state")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, "State: " + Bucket, t3 )
   Bucket = dw_4.getitemString(1, "loc_zip")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, "ZIP: " + Bucket )
   print(job, "")
   Bucket = dw_4.getitemString(1, "loc_phone")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, t1, "Phone: " + Bucket, t2 )
   //print(job, "")
   Bucket = dw_4.getitemString(1, "consent")
   if isNull(Bucket) then; Bucket = ""; end if
   print(job, "Consent: " + Bucket)
   print(job, "")

   printclose(job)

else
   messageBox("PRINT REPORT", "No printout available for this form.")
end if

return
```

```
//EXTENDS ANCESTOR SCRIPT
saveData()

string   newName, newSSN, testSSN   //containers for personal data
string   dwFilter                   //dataWindow filter string
string   bldg, rm, bunk          //dormitory assignment
string   Bucket                     //temp

int   msgCode     //return value of messageBox
int   dwCnt       //dataWindow count
int   checkCnt    //counts successful dataWindow updates
int   i, xx       //counters
long currRow      //current row indicator

dwCnt = upperBound(dwList[])

window otherSheet
userObject uoX
dataWindow dwX
dwItemStatus rowStat

setPointer( hourGlass!)

for i = 1 to dwCnt
   dwList[i].acceptText()
next

if (dw_1.dataObject = "d_regadmin1") AND (dw_1.modifiedCount()) > 0
then

   newSSN = dw_1.getItemString( 1, "ssnadmin")
   testSSN = dw_1.getItemString( 1, "ssnadmin", PRIMARY!, TRUE)

   if newSSN <> testSSN then

      Bucket = "mil.changeviewssn '" + testSSN + "', '"&
               + newSSN + "'"

      EXECUTE IMMEDIATE :Bucket;

      if sqlca.sqlcode < 0 then
         xx = messageBox("Data Entry Error",&
                         "Change of SSN failed.~n"&
                         + sqlca.SQLErrText + "~n~n"&
                         + "Restore original SSM?", question!,
YesNoCancel!, 1)
         choose case xx
            case 1
               dw_1.setFocus()
               dw_1.setColumn( "ssnadmin")
               dw_1.setText( testSSN)
               dw_1.acceptText()

            case 2
```

```
         //do nothing

      case else
         return
   end choose

end if

dw_1.setItemStatus( 1, "ssnadmin", Primary!, NotModified!)

//update SSN in other dws
rowStat = dw_2.getItemStatus( 1, 0, PRIMARY!)
if (dw_2.modifiedCount()) > 0 then
   w_admin.dw_2.setItem(1, "ssntarget", newSSN)
   dw_2.setItemStatus( 1, "ssntarget", Primary!,&
                          NotModified!)
else
   w_admin.dw_2.setItem(1, "ssntarget", newSSN)
   dw_2.setItemStatus( 1, 0, Primary!, NotModified!)
end if
rowStat = dw_2.getItemStatus( 1, 0, PRIMARY!)

rowStat = dw_3.getItemStatus( 1, 0, PRIMARY!)
if (dw_3.modifiedCount()) > 0 then
   w_admin.dw_3.setItem(1, "ssntarget", newSSN)
   dw_3.setItemStatus( 1, "ssntarget", Primary!,&
                          NotModified!)
else
   w_admin.dw_3.setItem(1, "ssntarget", newSSN)
   dw_3.setItemStatus( 1, 0, Primary!, NotModified!)
end if
rowStat = dw_3.getItemStatus( 1, 0, PRIMARY!)

rowStat = dw_4.getItemStatus( 1, 0, PRIMARY!)
if (dw_4.modifiedCount()) > 0 then
   w_admin.dw_4.setItem(1, "ssntarget", newSSN)
   dw_4.setItemStatus( 1, "ssntarget", Primary!,&
                          NotModified!)
else
   w_admin.dw_4.setItem(1, "ssntarget", newSSN)
   dw_4.setItemStatus( 1, 0, Primary!, NotModified!)
end if
rowStat = dw_4.getItemStatus( 1, 0, PRIMARY!)

//reset uo_1 in this and other sheets
Bucket = "ssn='" + testSSN + "'"
currRow = uo_1.dw_loco.find( Bucket, 1, 10000)
uo_1.dw_loco.setItem( currRow, "ssn", newSSN)

otherSheet = w_frame.getFirstSheet()
otherSheet = w_frame.getNextSheet(otherSheet)

do while isValid(otherSheet)
   for i = 1 to upperBound( otherSheet.control[] )
```

```
            if otherSheet.control[i].className() = "uo_1" then
               //assign to userObject variable,
               //so that control[] can be used
               uoX = otherSheet.control[i]

               for xx = 1 to upperBound( uoX.control[] )

                  if uoX.control[xx].className() = "dw_loco" then
                     //assign to dataWindow object,
                     //so that reset() can be used
                     dwX = uoX.control[xx]
                     dwX.reset()
                     xx = 100      //sentinel used to exit loop
                     i = 100  //sentinel used to exit loop
                  end if
               next
            end if
         next

         otherSheet = w_frame.getNextSheet(otherSheet)
      loop
   end if
end if

if (dw_2.modifiedCount()) > 0 then

   CHOOSE CASE dw_2.dataobject

   CASE "d_admin"
      w_admin.dw_2.setitem(1, "tdate", today())
      w_admin.dw_2.setitem(1, "uname", sqlca.userid)


   CASE "d_adminnew"

      dw_2.setColumn("n_student")
      newName = trim(dw_2.getText())

      if newName = "" then
         messageBox("Data Entry Error", &
                  "New record cannot be created without a name.", &
                   stopSign!)
         dw_2.setFocus()
         return
      end if

      //check if attempt to create new record
      //without unit at Company level
      dw_2.setColumn("unit")

      if (trim(dw_2.getText()) = "") AND (company <> "") then
         messageBox("Data Entry Error", &
                   "New record cannot be created without a Unit.",&
```

```
                    stopSign!)
            dw_2.setFocus()
            return
         end if

         dw_2.acceptText()
         dw_2.setColumn("ssntarget")
         newSSN = trim(dw_2.getText())
         if newSSN = ""  then
            messageBox("Data Entry Error", &
                       "New record cannot be created without SSN.", &
                        stopSign!)
            dw_2.setFocus()
            return
         end if

         if newSSN <> fetch_ssn then
            fetch_ssn = ""
         end if


         if w_admin.dw_1.modifiedCount() > 0 then
            w_admin.dw_1.setItem(1, "ssntarget", newSSN)
         end if
         if w_admin.dw_3.modifiedCount() > 0 then
            w_admin.dw_3.setItem(1, "ssntarget", newSSN)
         end if
         if w_admin.dw_4.modifiedCount() > 0 then
            w_admin.dw_4.setItem(1, "ssntarget", newSSN)
         end if

         fetch_ssn = newSSN

         w_admin.dw_2.setitem(1, "tdate", today())
         w_admin.dw_2.setitem(1, "uname", sqlca.userid)

      CASE "d_form90"
         if dw_6.rowCount() > 0 then
            for currRow = 1 to dw_6.rowCount()
               rowStat = dw_6.getItemStatus( currRow, 0, PRIMARY!)
               if (rowStat = newmodified!)
                  or (rowStat = datamodified!) then
                  dw_6.setitem( currRow, "d_tran", today())
                  dw_6.setitem( currRow, "n_user", sqlca.userid)
               end if
            next
            dw_6.update()
         end if

      CASE ELSE
         //do nothing

      END CHOOSE
   end if
```

```
if (dw_3.modifiedCount()) > 0 then
   w_admin.dw_3.setitem(1, "tdate", today())
   w_admin.dw_3.setitem(1, "uname", sqlca.userid)
end if

if ( dw_2.dataObject = "d_admin") AND ((dw_4.modifiedCount() > 0) OR
(dw_5.modifiedCount() > 0)) then
   Bucket = dw_4.getItemString( 1, "loc_qtrs")
   if ( Bucket = "B") OR ( Bucket = "BRKS") then
      if dw_5.modifiedCount() > 0 then
         bldg = dw_5.getItemString( 1, "bldg")
         rm = dw_5.getItemString( 1, "rm")
         bunk = dw_5.getItemString( 1, "bunk")

         if fetch_ssn = "" then
            fetch_ssn = dw_2.getItemString( 1, "ssnTarget")
         end if

         if isNull( bldg) OR isNull( rm) then
            messageBox("BARRACK DATA SAVE","You need to provide "&
                        + "at least Bldg and Room when "&
                        + "specifying location.")
            return
         end if

         if trim( bunk) = "" then
            setNull( bunk)
         end if
         msgCode = 0

         dwFilter = ""
         dw_6.setFilter( dwFilter)
         dw_6.filter()

         if dw_6.rowCount() = 0 then
            dw_6.retrieve( company)
         end if

         //check out from existing accomodation, if any
         Bucket = "brks_act_ssn=~"" + fetch_ssn + "~" "
         currRow = dw_6.find( Bucket, 1, dw_6.rowCount())
         if currRow > 0 then
            setNull( Bucket)
            dw_6.setItem( currRow, "brks_act_ssn", Bucket)
         end if

         //check if bldg and room exists
         dwFilter = "brks_act_bldg=~"" + bldg + "~" "&
                     + "and brks_act_rm=~"" + rm + "~""
         dw_6.setFilter( dwFilter)
         dw_6.filter()

         Bucket = ""
```

```
if dw_6.rowCount() = 0 then
   Bucket = "No such building and room is assigned "&
            + "to your Company.~n~n"&
            + "Do you want to drop your Building and "&
            + "Room assignment "&
            + "and save the rest of the data?"
else
   Bucket = "brks_act_bunk='" + bunk + "'"
   currRow = dw_6.find( Bucket, 1, 10)

   if currRow = 0 then
      Bucket = "No such bunk in Room " + rm + ".~n"&
            + "You can open Barracks Utilization window "&
            + "and add another bunk for this room. "&
            + "Then try the 'Save' operation again.~n~n"&
            + "Do you want to drop your Building and "&
            + "Room assignment "&
            + "and save the rest of the data?"
   else
      Bucket = ""
   end if
end if

if Bucket <> "" then
   msgCode = messageBox( "BARRACK DATA SAVE",&
                         Bucket, question!, yesNo!, 1)

   if msgCode = 2 then
      return
   end if
end if

if msgCode = 0 then
   //check if bunk is available
   if isNull( bunk) then
      //find 1st available bunk in the room
      Bucket = "isNull( brks_act_ssn)"
   else
      //check if given bunk in given room is empty
      Bucket = "isNull( brks_act_ssn) "&
               + "and brks_act_bunk='" + bunk + "'"
   end if

   currRow = dw_6.find( Bucket, 1, 10)

   if currRow = 0 then
      Bucket = "Room is already fully occupied. ~n~n" &
            + "Do you want to replace current occupant?"
      msgCode = messageBox( "BARRACK DATA SAVE",&
                            Bucket, question!, yesNo!, 2)

      if msgCode = 1 then
         if isNull( bunk) then
```

```
                  currRow = 1 //assign arbitrarily bunk A
                  bunk = "A"
               else
                  Bucket = "brks_act_bunk='" + bunk + "'"
                  currRow = dw_6.find( Bucket, 1, 10)

               end if
            else
               return
            end if
         else
            if isNull( bunk) then
               //assign first available bunk in room
               bunk = dw_6.getItemString( currRow,&
                                          "brks_act_bunk")
            end if
         end if

         Bucket = "not isNull( brks_act_ssn)"
         i = max( 1, dw_6.find( Bucket, 1, 10))
         Bucket = dw_6.getItemString( i, "admin_sex")

         if ( not isNull( Bucket)) &
            AND ( Bucket <> dw_2.getItemString( 1, "sex")) then
               messageBox( "BARRACK DATA SAVE",&
                           "Sex mismatch.~n~n"&
                           + "Open Barracks Utilization window "&
                           + "and resolve the discrepancy. "&
                           + "Then try the 'Save' "&
                           + "operation again.")
            return
         end if

         dw_6.setItem( currRow, "brks_act_ssn", fetch_ssn)
         dw_5.setItem( 1, "bunk", bunk)

         setNull( Bucket)
         dw_2.setItem( 1, "mealcard", fetch_ssn)
      end if
   end if
else
   //not living in barracks anymore => checkout
   dwFilter = ""
   dw_6.setFilter( dwFilter)
   dw_6.filter()

   if dw_6.rowCount() = 0 then
      dw_6.retrieve( company)
   end if

   Bucket = "brks_act_ssn=~"" + fetch_ssn + "~" "
   currRow = dw_6.find( Bucket, 1, dw_6.rowCount())
   if currRow > 0 then
      setNull( Bucket)
```

```
            dw_6.setItem( currRow, "brks_act_ssn", Bucket)
         end if
      end if
      if (dw_4.modifiedCount()) > 0 then
         dw_4.setItem( 1, "uname", sqlca.userID)
         dw_4.setItem( 1, "tdate", today())
      end if
   end if

   dw_5.resetUpdate()

   checkCnt = 0

   for i = 1 to dwCnt
      //if i <> 2 then //exclude dw_5
      if dwList[i].className() <> "dw_5" then
         checkCnt += dwList[i].update()
      else
         checkCnt += 1
      end if
   next

   if (checkCnt = dwCnt) then

      COMMIT using sqlca;

      if dw_2.dataobject = "d_adminnew" then
         uo_1.dw_plt.retrieve(company)
         uo_1.dw_loco.retrieve(company)
         uo_1.dw_loco.Filter()
      end if

      if newSSN <> fetch_ssn then
         fetch_ssn = newSSN
         //triggerevent( cb_fetch, clicked!)
         fetchData()
         primeNewRows( fetch_ssn)
      end if

   else
      messageBox("Data Entry Error", "Database update operation failed.",
   &
                 stopSign!)
      ROLLBACK using sqlca;
   end if

   if dw_6.visible=true then
      dw_6.bringToTop=true
   end if

   return
```

```
//PARAMETERS:  readonly string newSSN
//RETURN:      int 0 - success
//PURPOSE:     Inserts new row in selected dataWindows,
//             set the value of SSN to passed parameter,
//             set the row status to NEW!
int primeNweRows( readonly string newSSN)

if this.dw_2.rowcount() = 0 then
   this.dw_2.insertrow(0)
   this.dw_2.setitem(1, "ssntarget", newSsn)
   this.dw_2.setItemStatus(1, "ssntarget", PRIMARY!, dataModified!)
   this.dw_2.setItemStatus(1, "ssntarget", PRIMARY!, notModified!)
end if

if this.dw_3.rowcount() = 0 then
   this.dw_3.insertrow(0)
   this.dw_3.setitem(1, "ssntarget", newSsn)
   this.dw_3.setItemStatus(1, "ssntarget", PRIMARY!, dataModified!)
   this.dw_3.setItemStatus(1, "ssntarget", PRIMARY!, notModified!)
end if

if this.dw_4.rowcount() = 0 then
   this.dw_4.insertrow(0)
   this.dw_4.setitem(1, "ssntarget", newSsn)
   this.dw_4.setItemStatus(1, "ssntarget", PRIMARY!, dataModified!)
   this.dw_4.setItemStatus(1, "ssntarget", PRIMARY!, notModified!)
end if

return 0


//PARAMETERS:  readonly string formType
//RETURN:      int 0 - success
//PURPOSE:     Assigns dataWindow objects to dataWindow controls
//             depending on the formType
int setw_admin( string formType)

string selectedForm     //form type
string permList         //list of table permissions
string Bucket           //temp
int xx, dwCnt           //counters

triggerEvent(closeQuery!)

if ( this.title <> formtype) OR ( formtype = " Inprocessing ") then

   this.title = formtype
   selectedForm = upper(left(formtype, 4))

   if (fetch_ssn = "") AND (selectedForm <> "PT 1") then

      xx = messageBox("NEXT SOLDIER",&
                "Form Pt 2 and Pt 3 are not for new soldiers. ~n" &
              + "Do you wish to create a new record?",&
```

```
            Information!,& YesNo!, 1)

   if xx = 2 then
      return -1
   end if
end if

this.setRedraw( false)

dwCnt = upperBound( this.dwList)

if dwCnt = 0 then
   dwCnt = getDwList( this, dwList)
end if

this.dw_1.dataobject = "d_regadmin1"

choose case selectedForm

   case "PT 1"
      this.dw_2.dataobject = "d_admin"
      this.dw_3.dataobject = "d_depn"
      this.dw_4.dataobject = "d_local"
      this.dw_5.dataobject = "d_brk_loc"
      this.dw_6.dataobject = "d_bunk"

   case "PT 2"
      this.dw_2.dataobject = "d_form90"
      this.dw_3.dataobject = "d_trgdata"
      this.dw_4.dataobject = "d_office"
      this.dw_5.reset()
      this.dw_6.dataObject = "d_dlpt_hist"

   case "PT 3"
      this.dw_2.dataobject = "d_reserve"
      this.dw_3.dataobject = "d_comp"
      this.dw_4.dataobject = "d_sec"
      this.dw_5.reset()
      this.dw_6.reset()

   case else
      //messageBox("INPROCESSING", "Need to specify form.")
      this.uo_adminmenu.st_inprocessing.triggerEvent("lbuttonup")
      this.setRedraw(true)
      return 1
end choose

for xx = 1 to dwCnt
   dwList[xx].hide()
   dwList[xx].setTransObject(sqlca)
next

//enable/disable SAVE menu based on permissions
setSaveMenu()
```

```
    //find if user can create record for new soldier
    permList = space(4)
    sqlca.getSUID( "admin", sqlca.userID, permList)

    if pos( permList, "I") > 0 then
       this.cb_add.show()

       if fetch_ssn = "" then
          triggerEvent(this.cb_add, clicked!)
          this.setRedraw(true)
          return 0
       end if
    end if

    this.setRedraw( true)

    Bucket = "Enter/Edit Necessary Information and "&
             + "Select SAVE or Continue to "&
             + "Select Additional Information."

    w_frame.setMicroHelp( Bucket)
end if

return 0
```

8. Using Pipelines for Data Synchronization

PowerBuilder provides a feature, called data pipeline, which makes it possible to copy records from one or more source tables to a destination table. The destination table may already exists, or can be automatically created in the destination database at the time of data transfer. Data source and data destination can reside in the same database, or be in two separate databases managed by different DBMSs. Data pipes are applied in MILDB application for synchronization of data between remote local databases and the central database. There are five basic steps in data pipeline setup and execution:

- Step 1: Building supporting objects.
- Step 2: Connections setup.
- Step 3: Starting the pipeline and monitoring progress.

- Step 4: Handling row errors.
- Step 5: Closing pipeline.

a. Building Supporting Objects

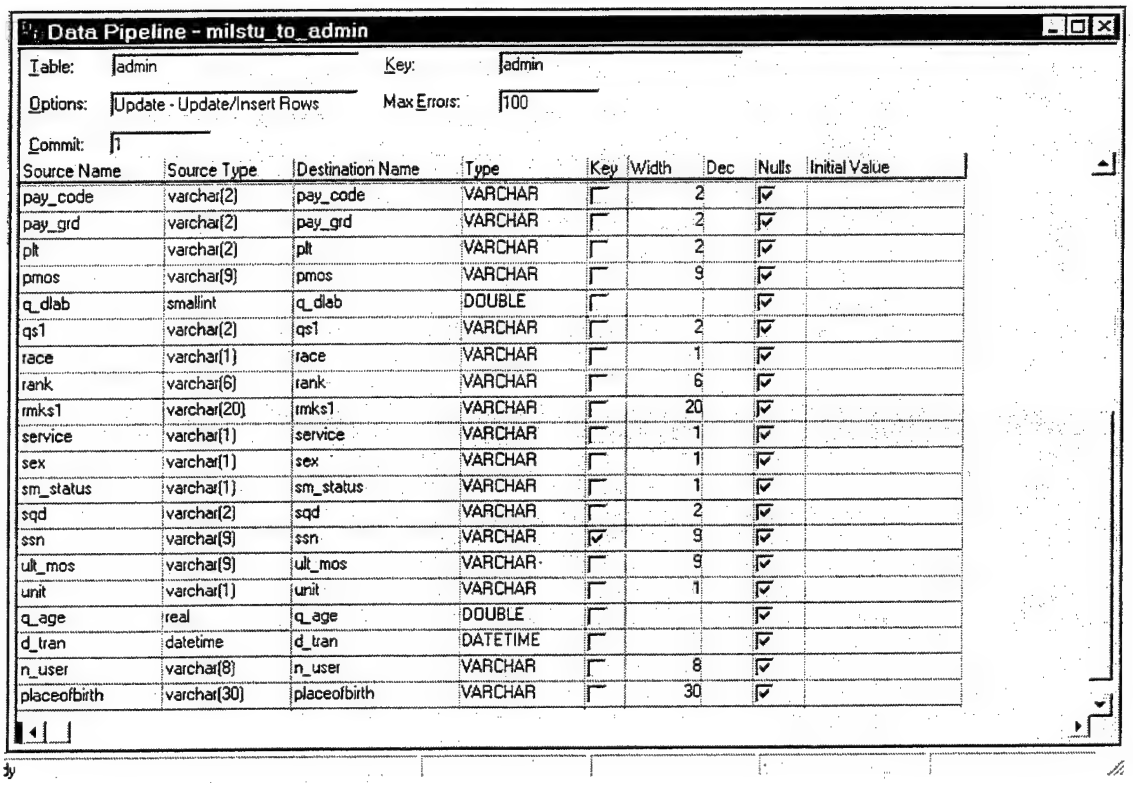
To implement a data pipeline, three objects are needed: Pipeline object, user object hosting the pipeline object, and a window for pipeline logistics.

(1) Pipeline Object

Pipeline object is created by means of PowerBuilder's object wizard. Pipeline object specifies:

- Source of data (one or more tables or views).
- Data destination (destination table or view).
- Type of piping operation (i.e., create new destination table and populate it with piped data; or: replace the contents of existing destination table with piped data; or: update data in destination table using piped data; or: append the destination table).
- Frequency of commits (indicates how often, after how many piped rows, should a COMMIT SQL statement be issued).
- Allowable number of errors (indicates number of errors that will be tolerated before the execution of data piping will be suspended. Error messages, along with the data, are captured in a dataWindow for further processing).
- Piping of extended attributes (indicates, whether extended attributes of the source data items are to be also piped to destination database).

Figure 17 shows an example of a pipeline object for piping data from central to local MILDB database.



Source Name	Source Type	Destination Name	Type	Key	Width	Dec	Nulls	Initial Value
pay_code	varchar(2)	pay_code	VARCHAR		2			
pay_grd	varchar(2)	pay_grd	VARCHAR		2			
plt	varchar(2)	plt	VARCHAR		2			
pmos	varchar(9)	pmos	VARCHAR		9			
q_dlab	smallint	q_dlab	DOUBLE					
qs1	varchar(2)	qs1	VARCHAR		2			
race	varchar(1)	race	VARCHAR		1			
rank	varchar(6)	rank	VARCHAR		6			
rmks1	varchar(20)	rmks1	VARCHAR		20			
service	varchar(1)	service	VARCHAR		1			
sex	varchar(1)	sex	VARCHAR		1			
sm_status	varchar(1)	sm_status	VARCHAR		1			
sqd	varchar(2)	sqd	VARCHAR		2			
ssn	varchar(9)	ssn	VARCHAR		9			
ult_mos	varchar(9)	ult_mos	VARCHAR		9			
unit	varchar(1)	unit	VARCHAR		1			
q_age	real	q_age	DOUBLE					
d_tran	datetime	d_tran	DATETIME					
n_user	varchar(8)	n_user	VARCHAR		8			
placeofbirth	varchar(30)	placeofbirth	VARCHAR		30			

Figure 17. Pipeline Object milstu_to_admin

(2) Supporting User Object

Pipeline object is similar to dataWindow object. It contains selection of columns from both the source and destination tables, but the object has no properties, events, or functions. To acquire these, the pipeline object needs a host object. PowerBuilder provides for this purpose a special pipeline system object, which contains properties, events, and functions needed for pipeline operations.

Pipeline properties include:

- Data Object (name of pipeline object).
- RowsRead (cumulative number of rows read since the pipeline started).
- RowsWritten (cumulative number of rows written to destination table).
- RowsInError (number of rows rejected by the destination database).

Pipeline events include:

- PipeStart (triggered when pipeline starts).
- PipeMeter (triggered after every COMMIT command).
- PipeEnd (triggered when pipeline finishes execution, or is stopped).

Pipeline functions include:

- Start (starts the pipeline).
- Repair (attempts to write corrected, previously rejected, rows to the destination database).
- Cancel (cancels pipeline execution).

The following code shows scripts from non-visual object o_synchro_pipe:

```
//USER OBJECT O_SYNCHRO_PIPE

//INSTANCE VARIABLES
staticText statusRead, statusWritten, statusError //for pipe status
data
staticText flowAnimation          //for pipe flow animation

string arrowText                  //text of flowAnimation
boolean toLeft                    //indicator of direction of animation
char arrowChar                    // char '<' or '>' for anim


//EVENT PIPESTART
arrowtext = flowAnimation.text

//set the char to be added to arrowtext to achieve animation effect
if pos( arrowtext, "<") > 0 then
   toLeft = true
   arrowChar = "<"
else
   toLeft = false
   arrowChar = ">"
end if


//EVENT PIPEMETER
statusRead.text = string( rowsRead)
statusWritten.text = string( rowsWritten)
statusError.text = "Rows in error~r" + string( rowsInError)

long posOut       //position of char to be removed from arrowText

char charIn       //adds char to pipe movement animation

if rowsInError = 100 then
   messageBox( "DATA PIPE", "100 rows have one or more data items "&
               + "that were rejected "&
               + "by destination database. ~n~n"&
               + "Click 'Apply Known Fixes' and 'Continue', "&
               + "or correct the errors "&
               + "manually and click 'Continue'.")
end if

//to create animation effect of moving errows:
//at given interval of rowsWritten add space or ">" at one end of
string
//and remove one char at the other end
if rowsRead = rowsWritten then
   if mod( rowsWritten, 4) = 0 then
      charIn = arrowChar
   else
      charIn = " "
   end if
```

```
    if toLeft = true then
       arrowText = arrowText + charIn
       posOut = 1
    else
       arrowText = charIn + arrowText
       posOut = len( arrowText)
    end if

    arrowText = replace( arrowText, posOut, 1, "")
end if

flowAnimation.text = arrowText
```

(3) Window

Window that will provide logistics for a pipeline needs to contain the following objects:

- DataWindow control (pipeline will insert into this dataWindow any row in error. Later, these rows can be corrected and an attempt can be made to write them to the destination database by calling a function repair()).
- Optional command buttons for pipeline start, repair, or cancel.
- Optional text field for displaying the number of rows read, written, and rows in error.
- Other optional controls, informing the user about the pipeline progress, such as direction of data flow, etc.

The pipeline window in MILDB application provides numerous features that allow a successful execution data synchronization even by an occasional user, such as:

- Visual selection of data source and destination.
- Visual selection of source tables to be synchronized.

- Animated indicator of data flow.
- Pipeline start/stop button.
- Display of the number of rows read, written, and in error.
- Button for applying fixes of the most common data errors.
- Button for resuming the pipeline operation after applying the fixes.
- Button for suspension of pipeline operation.

Figure 18 shows the design of a window supporting pipeline operation.

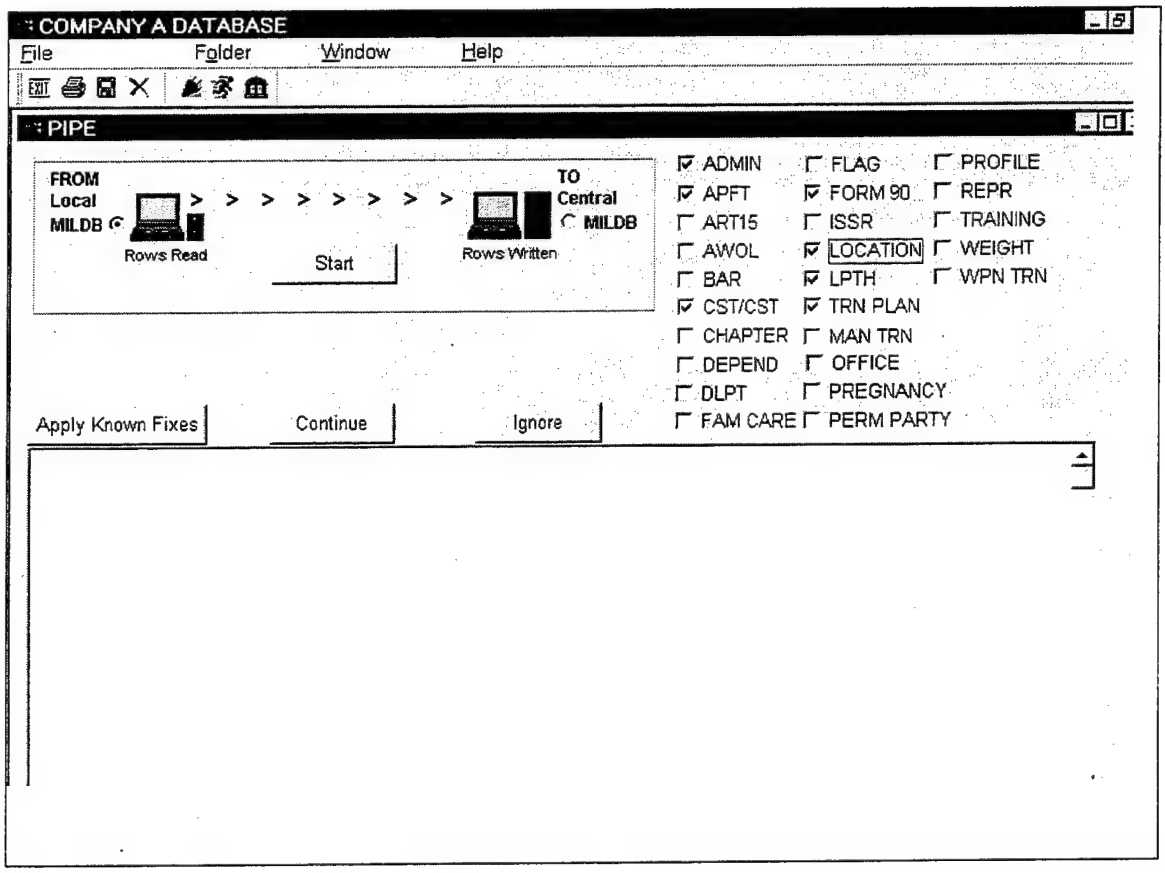


Figure 18. Window w_pipe

b. Connections Setup

In order to prepare the window for pipeline execution, the following tasks need to be accomplished:

- Establish connection with the source and destination database:

```
//transaction object for source DATABASSE
transaction sourceConnection
sourceConnection = CREATE transaction

//transaction object for destination database
transaction destinationConnection
destinationConnection = CREATE transaction

//ODBC connection setup
sourceConnectString = "ConnectString='DSN=MILX';
destConnectString = "ConnectString='DSN=milstu'

//connect to databases
CONNECT using sourceConnection;
CONNECT using destinationConnection;
```

- Instantiate the host user object:

```
//pipe user object
o_synchro_pipe pipeLogistics
pipeLogistics = CREATE o_synchro_pipe
```

- Assign the pipeline object:

```
pipeLogistics.dataObject = <pipeObjectName>
```

c. Starting Pipeline and Monitoring Progress

The basic syntax for starting the pipeline execution is:

<[int] return code> = *<pipeline userObject>* . Start(*<source Trans. Object>*, &
<destination Trans. Object> , &
<name of dataWindow for storing errors>)

The return code indicates successful start of pipeline, or possible causes of failure. The following code shows scripts for database connections, setup of the pipeline, and starting the pipeline, contained in window w_pipe:

```
//***WINDOW W_PIPE

//INSTANCE VARIABLES
transaction sourceConnection          //transaction object
                                      //for source DATABASE

transaction destinationConnection     //transaction object
                                      //for destination database

o_synchro_pipe pipeLogistics          //pipe user object

checkBox taskBoxList[]                //array of controlBoxes

boolean hideHelp                      //hide help true/false
```

```
//WINDOW EVENTS

//WINDOW OPEN EVENT

//Local variables
int xx, yy     //step counters

disconnect;

sourceConnection = CREATE transaction
destinationConnection = CREATE transaction

//instantiate pipe object for pipe logistics
pipeLogistics = CREATE o_synchro_pipe


//link window status text objects with text object of pipeLogistics
pipeLogistics.statusRead = st_rowsread
pipeLogistics.statusWritten = st_rowswritten
pipeLogistics.statusError = st_rowsinerror
pipeLogistics.flowAnimation = st_pipeFlow

//initialize list of checkBoxes
yy = 1
for xx = 1 to upperBound(this.control[])
   if this.control[xx].typeOf() = checkBox! then
      taskBoxList[yy] = this.control[xx]
      yy++
   end if
next



//EVENT MOUSEMOVE
if hideHelp = true then
   st_start.hide()
   st_fix.hide()
   st_continue.hide()
   st_ignore.hide()
   hideHelp = false
end if


//WINDOW FUNCTIONS

//PARAMETERS:  string boxTxt - text label of checkBox control
//RETURN:      boolean true/false
//PURPOSE:     Indicates whether checkBox is checked
isChecked( string boxTxt)

int xx   //counter

for xx = 1 to upperBound( taskBoxList[])
   if lower( taskBoxList[xx].text) = lower( boxTxt) then
```

```
      if taskBoxList[xx].checked = true then
         return TRUE
      else
         xx = 1000
      end if
   end if
next

return FALSE


//EVENTS OF CONTROLS

//BUTTON cb_start
//CLICKED EVENT

string sourceDsn, destDsn                              //data source names
string sourceConnectString, destConnectString          //connection strings
string pipeObjectName                                  //name of pipe object
string pipeList[]                                      //array of pipes
string Bucket                                          //temp

int xx                                                 //step counter
int startFlag                                          //pipe start flag

long currRow, maxRow
long posX

pipeList = { "admin", "apft","art15", "awol", "bar", "cc_trng",&
             "chap", "depn", "dlpt", "fcare", "flag", "form90",&
             "issr", "loc", "lpth", "lrt_plan", "m_trng","office",&
             "preg", "prm_prty", "prof", "repr", "trng", "wc",&
             "wpn"}


if this.text = "Start" then

   if rb_milx.checked = true then

      //data transfer from MILX to MILSTU
      sourceDsn = "MILX"
      destDsn = "MILSTU"

      sourceConnectString = "ConnectString='DSN=MILX'; "&
                            + "Time=' ''''hh:mm:ss:'''' ';"&
                            + "delimitidentifier='NO';"&
                            + " MsgTerse='Yes'"

      destConnectString = "ConnectString='DSN=milstu';"&
                          + " MsgTerse='Yes'"

   else
      //data transfer from MILSTU to MILX
```

```
    sourceDsn = "MILSTU"
    destDsn = "MILX"

    sourceConnectString = "ConnectString='DSN=milstu';"&
                            + " MsgTerse='Yes'"

    destConnectString = "ConnectString='DSN=MILX'; "&
                          + "Time=' ''''hh:mm:ss:''' ';"&
                          + "delimitidentifier='NO';"&
                          + " MsgTerse='Yes'"
end if

//set properties of transaction objects
sourceConnection.dbms = "ODBC"
destinationConnection.dbms = "ODBC"

sourceConnection.dbParm = sourceConnectString
destinationConnection.dbParm = destConnectString

//// Profile milstu_native
//SQLCA.DBMS = "MSS MS Microsoft SQL Server 6.x"
//SQLCA.Database = "mildb"
//SQLCA.LogPass = "mil"
//SQLCA.ServerName = "pomdb"
//SQLCA.LogId = "mil"
//SQLCA.AutoCommit = False
//SQLCA.DBParm = ""


//connect to source and destination databases
disconnect;
CONNECT USING sourceConnection;

if sourceConnection.sqlcode < 0 then
   messageBox("DATABASE CONNECT","Could not connect to "&
                + sourceDsn + ".", exclamation!)
   return
end if

CONNECT USING destinationConnection;

if destinationConnection.sqlcode < 0 then
   messageBox("DATABASE CONNECT","Could not connect to "&
                + destDsn + ".", exclamation!)
   DISCONNECT USING sourceConnection;
   return
end if

this.text = "Stop"

//set pipe object
for xx = 1 to upperBound( pipeList[])
   //check if table to pipe selected
   if isChecked( pipeList[xx]) = true then
```

```
    //get pipe object name
    if rb_milx.checked = true then
       if pipeList[xx] = "dlpt" then
          continue
       else
          pipeObjectName = "milx_to_" + pipeList[xx]
       end if
    else
       pipeObjectName = "milstu_to_" + pipeList[xx]
    end if

    pipeLogistics.dataObject = pipeObjectName

    //***start pipe
    startFlag = pipeLogistics.Start( sourceConnection,&
                destinationConnection, dw_pipe_errors, Company)

    Bucket = ""

    choose case startFlag
       case -1
          Bucket = "Pipe open failed."

       case -5
          Bucket = "Missing connection."

       case -15
          Bucket = "Pipe already in progres."

       case -16
          Bucket = "Error in source database."

       case -17
          Bucket = "Error in destination database."
    end choose

    if Bucket <> "" then
       messageBox( "PIPE ERROR", Bucket&
                   + "~n~nOperation halted.", exclamation!)
       return
    end if
end if

maxRow = dw_pipe_errors.rowCount()

if maxRow > 0 then
    for currRow = 1 to maxRow
       Bucket = dw_pipe_errors.getItemString( currRow, 1)
       posX = pos( Bucket, ":")

       if posX > 0 then
          Bucket = mid( Bucket, posX + 1 )
          dw_pipe_errors.setItem( currRow, 1, Bucket)
       end if
```

```
            next
         end if

      next


      DISCONNECT USING sourceConnection;
      DISCONNECT USING destinationConnection;

      this.text = "Start"

   else
      //call the Cancel function of pipe object
      if pipeLogistics.Cancel() = 1 then
         Beep(1)
         this.text = "Start"
      else
         messageBox( "PIPE ERROR", "Error while trying to stop data
   transfer.", exclamation!)
      end if
   end if



   //EVENT MOUSEMOVE
   if this.text = "Start" then
      st_start.text = "Start piping data from selected tables"
   else
      st_start.text = "Stop piping data"
   end if
   st_start.show()
   hideHelp = true


   //RADIO BUTTON rb_milx
   //EVENT CLICKED
   st_milx.text = "FROM~rLocal"
   st_milstu.text = "TO~rCentral"
   st_pipeFlow.rightToLeft = false
   st_pipeFlow.text = " >    >    >    >    >    >    >    >    >    >"
   st_milxrows.text = "Rows Read"
   st_milstuRows.text = "Rows Written"
   st_rowsread.text = ""
   st_rowswritten.text = ""
   st_rowsinerror.text = ""
   st_rowsread.X = st_milxrows.X
   st_rowswritten.X = st_milstuRows.X

   dw_pipe_errors.reset()



   //RADIO BUTTON rb_STU
```

```
//EVENT CLICKED
st_milx.text = "TO~rLocal"
st_milstu.text = "FROM~rCentral"
st_pipeFlow.rightToLeft = true
st_pipeFlow.text = "<    <    <    <    <    <    <    <    <    < "
st_milxrows.text = "Rows Written"
st_milstuRows.text = "Rows Read"
st_rowsread.text = ""
st_rowswritten.text = ""
st_rowsinerror.text = ""
st_rowsread.X = st_milstuRows.X
st_rowswritten.X = st_milxrows.X

dw_pipe_errors.reset()



//BUTTON cb_continue
//CLICKED EVENT
if pipeLogistics.repair( destinationConnection) <> 1 then
   messageBox( "PIPE ERROR", "Error when trying to apply"&
               + " fixes.~n~n"&
               + "Check you data or choose to ignore rows "&
               + with errors.",&
               exclamation!)
end if


//EVENT MOUSEMOVE
st_continue.show()
hideHelp = true


//BUTTON cb_ignore
//CLICKED EVENT
dw_pipe_errors.reset()

if pipeLogistics.repair( destinationConnection) <> 1 then
   messageBox( "PIPE ERROR", "Error when trying to apply fixes.",&
               exclamation!)
end if


//EVENT MOUSEMOVE
st_ignore.show()
hideHelp = true
```

d. Handling Row Errors

When a pipeline is unable to write particular rows to the destination table due to some errors (i.e., violation of the primary key, violation of referential integrity, etc.), these rows are inserted into the pipeline error dataWindow. When the number of rows in error reaches the maximum indicated in the pipeline object, execution of data piping is suspended. The user has an option to discard rows in error and resume the pipeline operation, or correct the data and attempt to write them to the destination database by calling the function:

Repair(<*destination trans. object*>)

Before the pipeline can resume its normal operation, repaired rows that were written to the destination database have to be committed by statement

COMMIT using <*destination trans. object*> ;

e. Closing Pipeline

When the data transfer is concluded, there is no need to explicitly destroy objects that were dynamically created in preparation for pipeline execution. PowerBuilder's garbage collection mechanism will remove these objects automatically after they seize to be referenced in scripts. Good programming practice still calls for disconnecting the application from both the source and destination database, using commands:

DISCONNECT using <*source trans. object*> ;

DISCONNECT using <*destination trans. object*> ;

9. Running the MILDB Application

When a user starts the MILDB application, he/she is challenged by the database authentication procedure. After passing the authentication test, a window frame with the main menu bar opens. The user can choose from three major areas of operation listed in the Folder menu:

- Administrative (represented by a 'pencil' icon on the menu bar).
- Physical Training & Weight Control (represented by a 'running man' icon on the menu bar).
- Dormitory Room Assignment (represented by a 'building' icon on the menu bar).

Each selection opens a separate window which is a gateway to these distinct areas of operation. The Administrative part of the application serves for creating new student record, and for viewing and editing student biographical and administrative data. The Physical Training & Weight Control part of the application allows to create, edit, and query data related to physical training and weight control. The Dormitory Room Assignment portion of the application provides an interface for assigning students to dormitory rooms. Whenever a selection of a person from a list of personnel is needed before any data can be retrieved, the Locator appears automatically on the screen. By clicking a name of a person, retrieval and display of data is triggered. Anytime the type of a report needs to be determined before data can be retrieved, a popup menu prompts the user for selection from a list of reports and forms.

User can see only names and records that he/she is authorized to access. Records can be edited by clicking on selected data item and typing new value. Navigation between fields can be achieved by pressing the TAB or ENTER key. Saving the data can be triggered by clicking the 'Save' icon in menu bar, or by selecting 'Save' from the main menu. The 'Save' feature is automatically enabled/disabled, depending on user's privileges. Displayed data can be sent to a printer, or exported to a text file. Both features are available in the File menu on the main menu.

Global users, who have access to data from more than one Unit, see slightly modified main menu with added capabilities. Their menu contains an icon for every Unit in the database. Global user can freely switch from one Unit to another. After each switch, names in the Locator are automatically replaced by names from selected Unit.

The MILDB application, in its final version, is a multifaceted application with a wealth of features. It has been noted, however, that by organizing the application's interface into logical groups, and by providing visual guidance and clues to users during each action, it is easy to use. Generally, only about 20 minute briefing is need for a new user to become proficient in using all major features of MILDB.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

Implementation of a two-tier client/server model, proposed in this thesis, provides only an interim solution. Growing demands for data exchange will soon command implementation of a new, fault-tolerant system that will provide faster response and easier, yet secure, access to information. Next research should, therefore, focus on the development of a client/server model that will be built on an open systems foundation and will meet these demands, allowing to integrate new client/server software systems and various middleware and application standards as they emerge. The resulting client/server model should ensure continuous systems' interoperability, scalability, and portability in the heterogeneous computing environment at the Presidio of Monterey.

THIS PAGE INTENTIONALLY LEFT BLANK

BIBLIOGRAPHY

1. Harrington, Jan L., *Relational Database Management for Microcomputers: Design and Implementation,* Holt, Rinehart and Winston, Inc., 1988.

2. Stonebreaker, Michael, *Object-Relational DBMSs: The next Name,* Morgan Kaufmann Publishing, Inc., 1996.

3. Elmasri, R., Navathe, S., *Fundamentals of database Systems,* The Benjamnin/Cummings Publishing Company, Inc., 1994.

4. Gruber, Martin, *Understanding SQL,* Sybex, 1990.

5. Celko's, Joe, *SQL for Smarties: Enhanced SQL Programming,* Morgan Kaufmann Publishers, Inc., 1995.

6. Wille, Christoph, et. al., *MCSE SQL Server 7 Administration,* New Riders Publishing, 1999.

7. Microsoft, *SQL Server 7.0 System Administration Training,* Microsoft Press, 1999.

8. Otey, Michael, Conte, Paul, *SQL Server 7 Developer's Guide,* Osborne McGraw Hill, 1999.

9. Jennings, Roger, *Using Access 97,* Que, 1997.

10. Gourgani, Kouros, *Enterprise Components and PowerBuilder 7,* The Romo Technology Group, 1999.

11. Rennhackkamp, Martin, *An Analysis of the Strengths and Weaknesses of the Big Six Database Servers,* DBMS Online, http://www.dbmsmag.com/9611d52.html, 1996.

12. Gupta, Gopal, *The relational Data Model,* http://www.cs.jcu.edu.au/ftp/web/teaching/Subjects/cp3020/1997/Lecture_Notes/relational_Model, 1996.

13. Sybase, *Installation Guide,* Sybase, 1999.

14. Sybase, *Building Internet and Enterprise Applications,* Sybase, 1999.

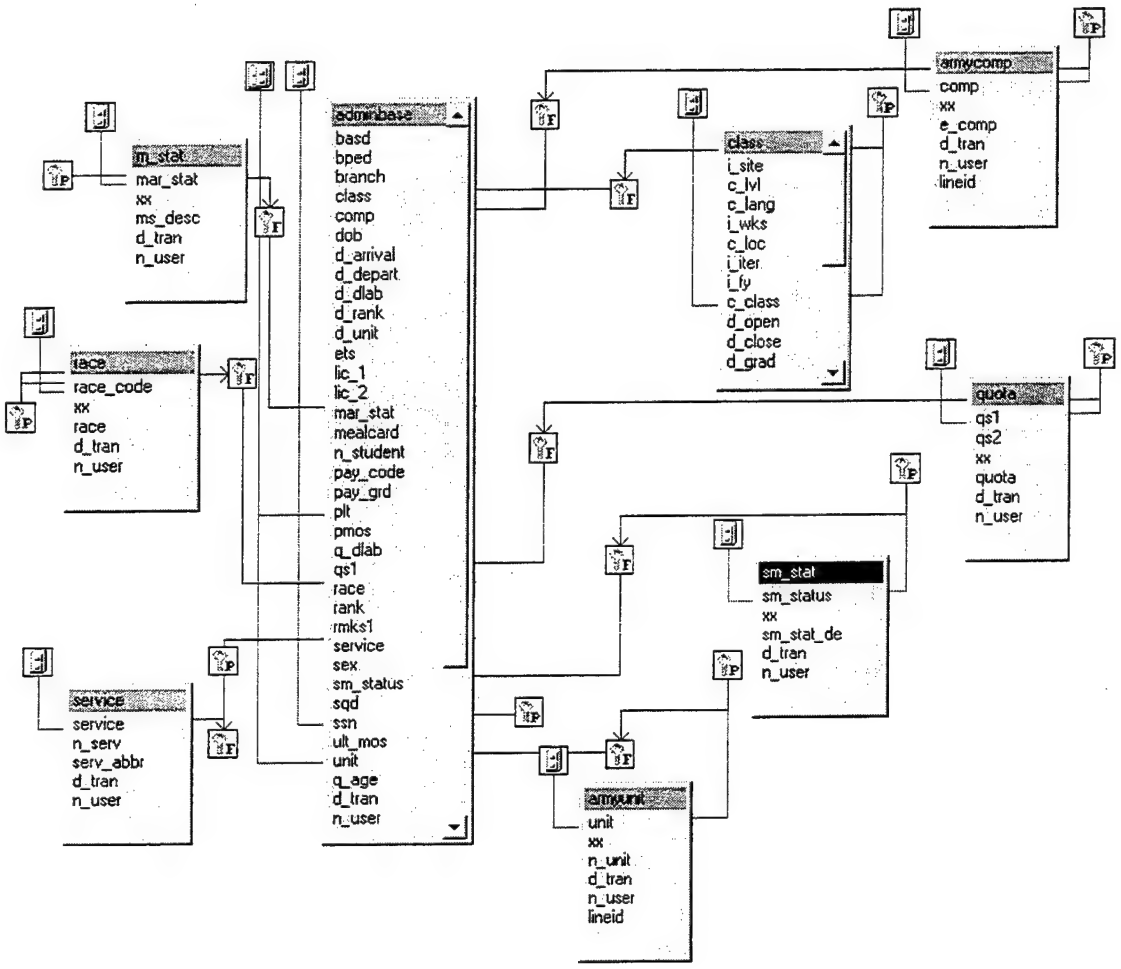
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A

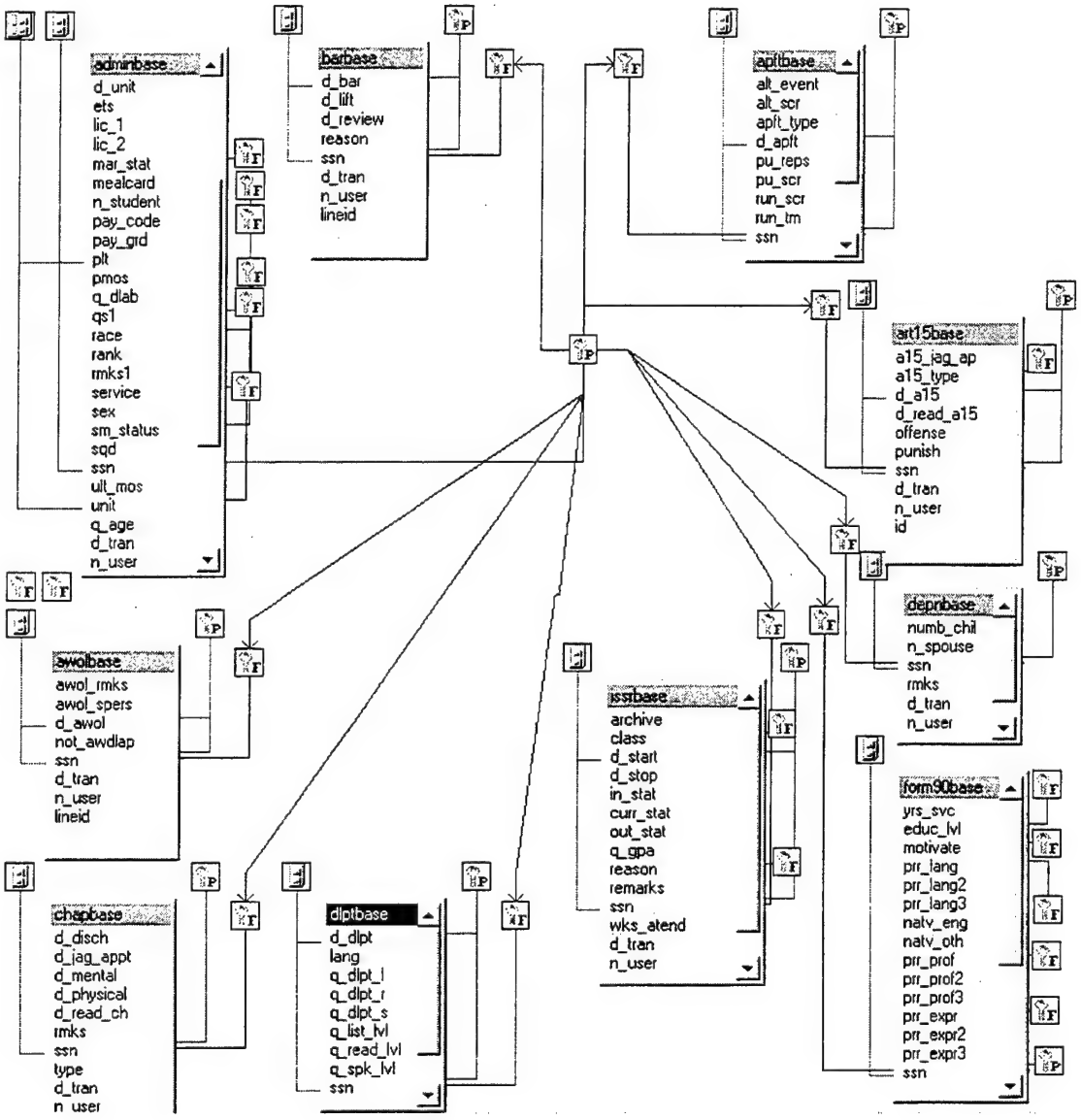
Documentation of MILDB Database schema includes:

- ADMINBASE table with reference tables
- Tables depending on ADMINBASE
- Reference tables
- Views

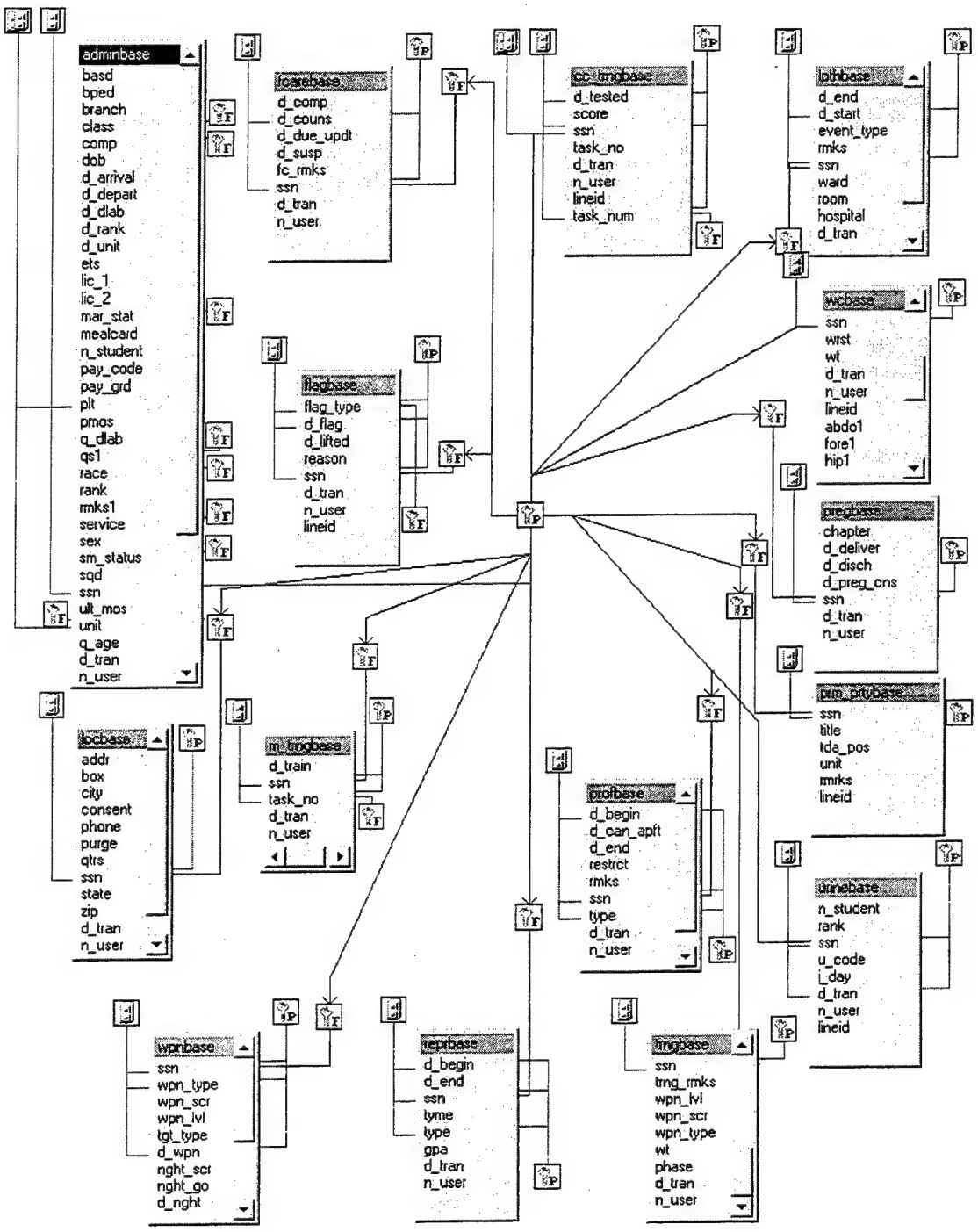
Table ADMINBASE with Reference Tables:



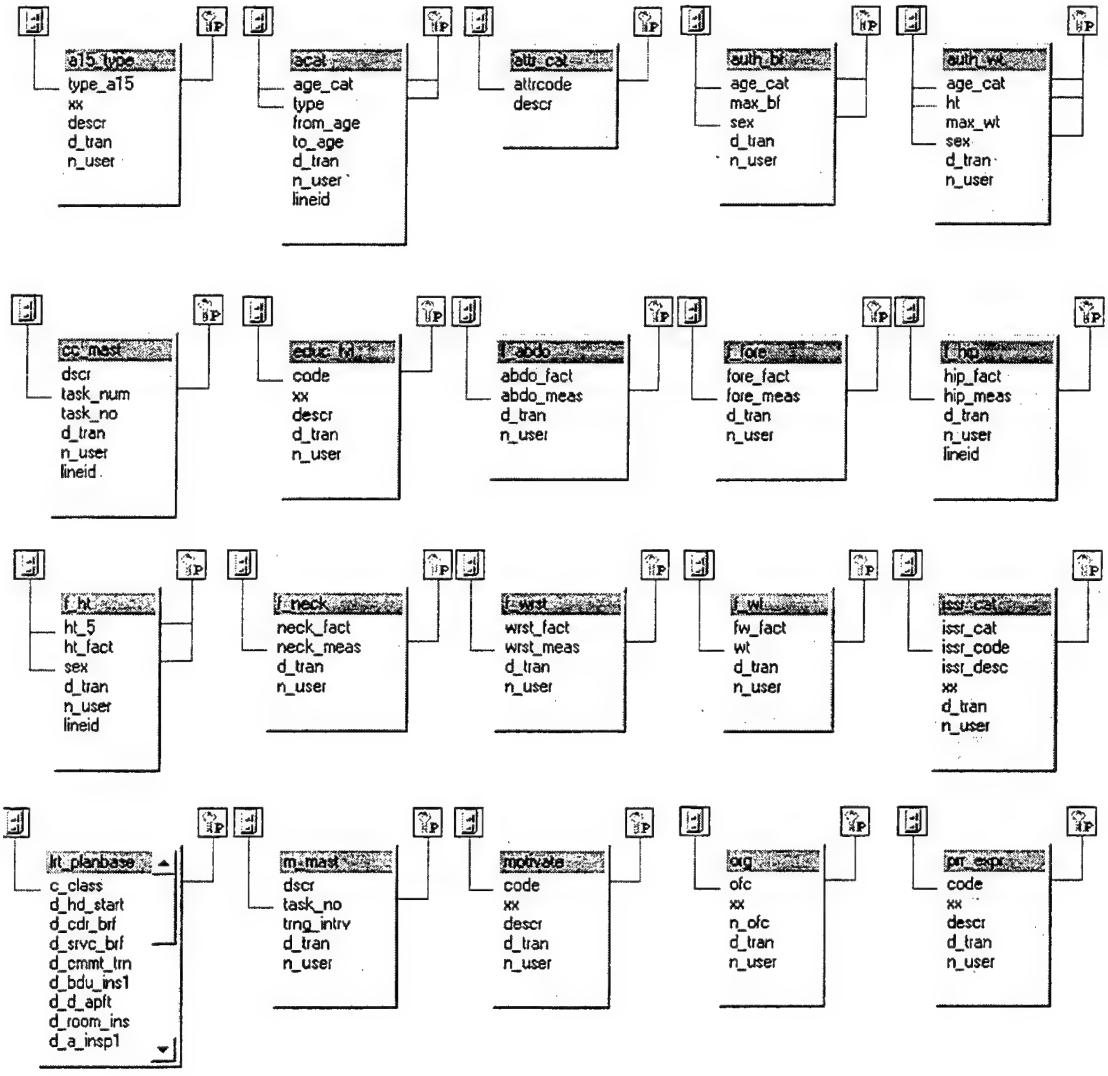
Tables depending on ADMINBASE:



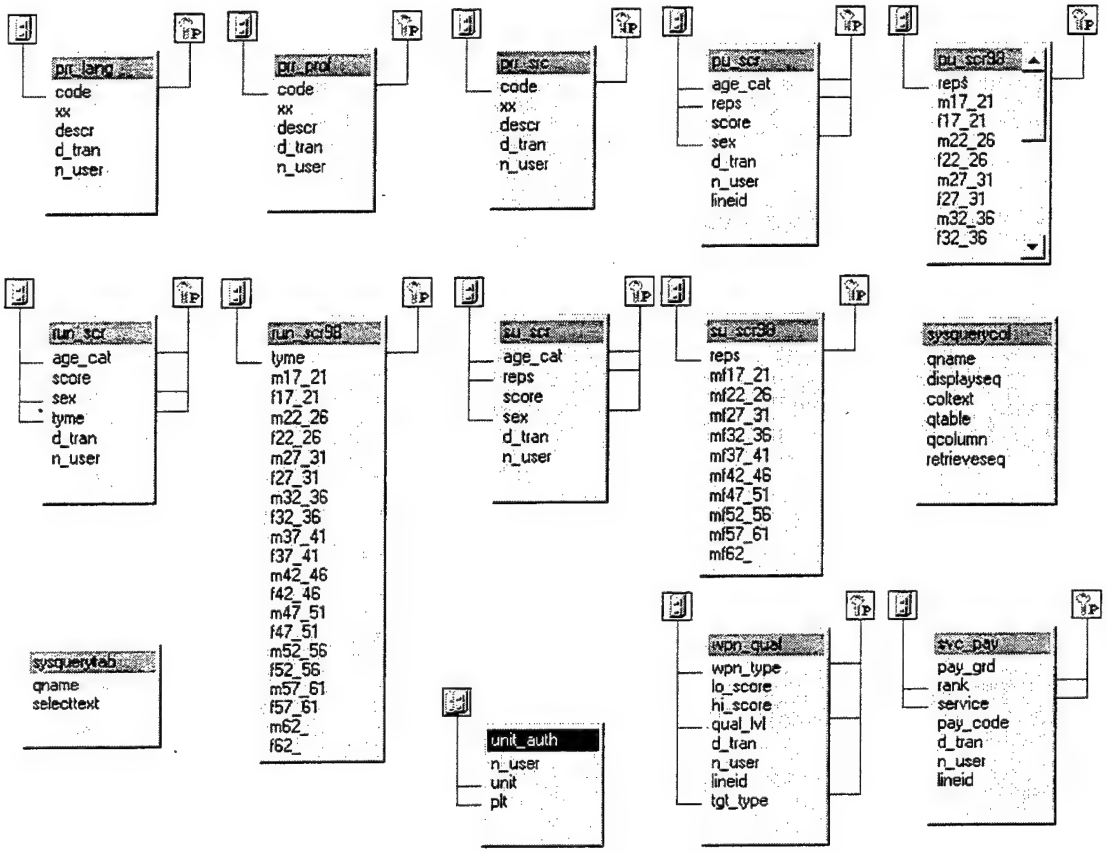
Tables depending on ADMINBASE (continued):



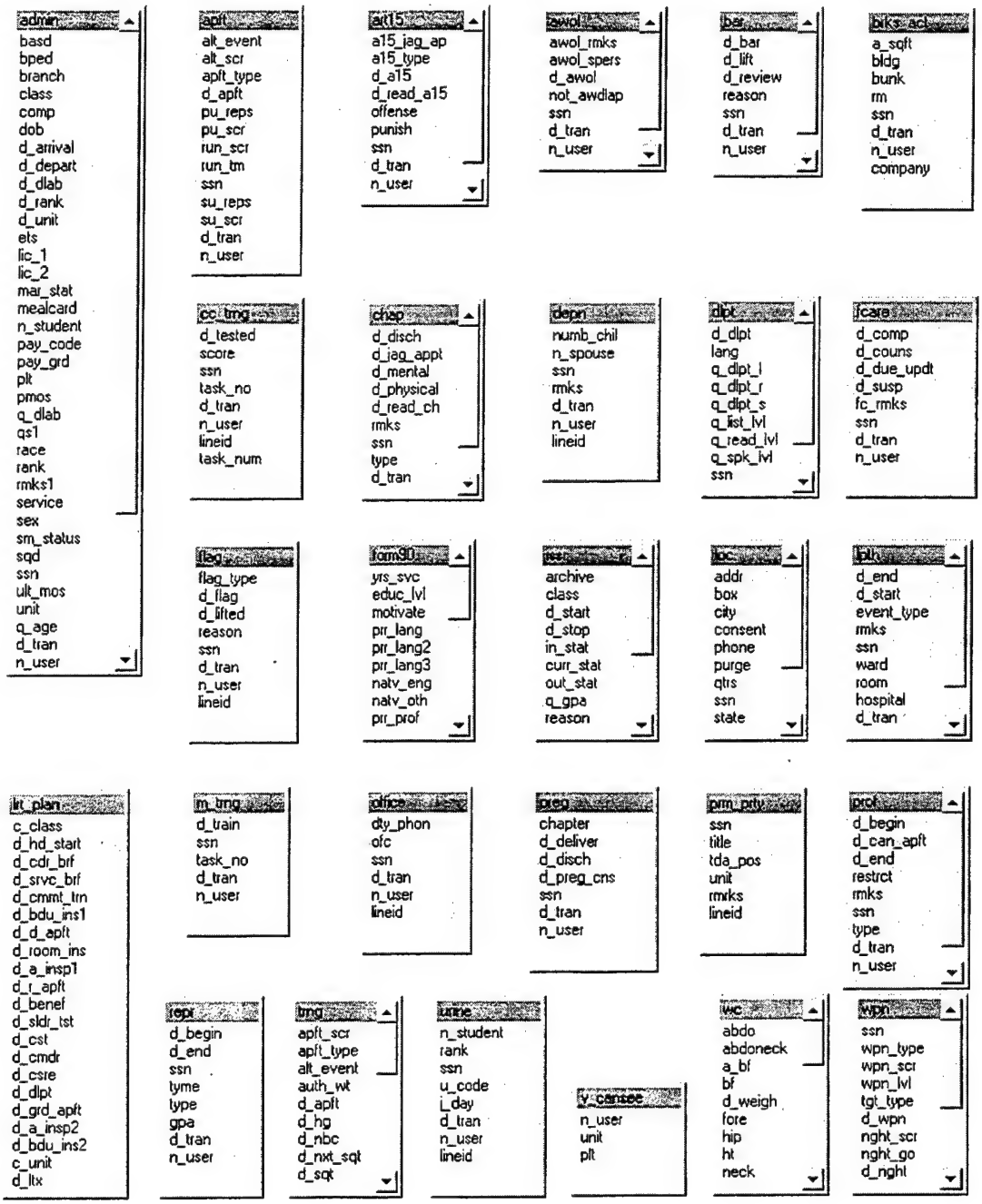
MILDB Reference Tables:



MILDB Reference Tables (continued):



MILDB Views:


admin
basd
bped
branch
class
comp
dob
d_arrival
d_depart
d_dlab
d_rank
d_unit
ets
lic_1
lic_2
mar_stat
mealcard
n_student
pay_code
pay_grd
plt
pmos
q_dlab
qs1
race
rank
rmks1
service
sex
sm_status
sqd
ssn
ulk_mos
unit
q_age
d_tran
n_user
apft
alt_event
alt_scr
apft_type
d_apft
pu_reps
pu_scr
run_scr
run_tm
ssn
su_reps
su_scr
d_tran
n_user
art15
a15_jag_ap
a15_type
d_a15
d_read_a15
offense
punish
ssn
d_tran
n_user
awol
awol_rmks
awol_spers
d_awol
not_awdlap
ssn
d_tran
n_user
bar
d_bar
d_lift
d_review
reason
ssn
d_tran
n_user
bks_acl
a_sqft
bldg
bunk
rm
ssn
d_tran
n_user
company
cc_trng
d_tested
score
ssn
task_no
d_tran
n_user
lineid
task_num
chap
d_disch
d_jag_appt
d_mental
d_physical
d_read_ch
rmks
ssn
type
d_tran
depn
numb_chil
n_spouse
ssn
rmks
d_tran
n_user
lineid
dlpt
d_dlpt
lang
q_dlpt_l
q_dlpt_r
q_dlpt_s
q_list_lvl
q_read_lvl
q_spk_lvl
ssn
fcare
d_comp
d_couns
d_due_updt
d_susp
fc_rmks
ssn
d_tran
n_user
flag
flag_type
d_flag
d_lifted
reason
ssn
d_tran
n_user
lineid
form90
yrs_svc
educ_lvl
motivate
pr_lang
pr_lang2
pr_lang3
natv_eng
natv_oth
pr_prof
archive
class
d_start
d_stop
in_stat
curr_stat
out_stat
q_gpa
reason
loc
addr
box
city
consent
phone
purge
qtrs
ssn
state
lpb
d_end
d_start
event_type
rmks
ssn
ward
room
hospital
d_tran
lt_plan
c_class
d_hd_start
d_cdr_brf
d_srvc_brf
d_cmmt_trn
d_bdu_ins1
d_d_apft
d_room_ins
d_a_insp1
d_r_apft
d_benef
d_sldr_tst
d_cst
d_cmdr
d_csre
d_dlpt
d_grd_apft
d_a_insp2
d_bdu_ins2
c_unit
d_ltx
m_trng
d_train
ssn
task_no
d_tran
n_user
ofrce
dty_phon
ofc
ssn
d_tran
n_user
lineid
preg
chapter
d_deliver
d_disch
d_preg_cns
ssn
d_tran
n_user
prm_prty
ssn
title
tda_pos
unit
rmks
lineid
prof
d_begin
d_can_apft
d_end
restrct
rmks
ssn
type
d_tran
n_user
rspt
d_begin
d_end
ssn
tyme
type
gpa
d_tran
n_user
trng
apft_scr
apft_type
alt_event
auth_wt
d_apft
d_hg
d_nbc
d_nxt_sqt
d_sqt
unme
n_student
rank
ssn
u_code
i_day
d_tran
n_user
lineid
v_cansee
n_user
unit
plt
wc
abdo
abdoneck
a_bf
bf
d_weigh
fore
hip
ht
neck
wpn
ssn
wpn_type
wpn_scr
wpn_lvl
tgt_type
d_wpn
nght_scr
nght_go
d_nght

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

Documentation of MILDB table and view definition includes:

- Definition of table ADMINBASE
- Definition of tables referenced by ADMINBASE
- Definition of views
- Definition of stored procedures

```
CREATE TABLE [mil].[CLASS] (
    [I_SITE] [varchar] (3) NULL ,
    [C_LVL] [varchar] (2) NULL ,
    [C_LANG] [varchar] (2) NULL ,
    [I_WKS] [varchar] (2) NULL ,
    [C_LOC] [varchar] (1) NULL ,
    [I_ITER] [varchar] (3) NULL ,
    [I_FY] [varchar] (2) NULL ,
    [C_CLASS] [varchar] (12) NOT NULL ,
    [D_OPEN] [datetime] NULL ,
    [D_CLOSE] [datetime] NULL ,
    [D_GRAD] [datetime] NULL ,
    [D_RPT] [datetime] NULL ,
    [Q_ORIG_SCH] [float] NULL ,
    [Q_CURR_SCH] [float] NULL ,
    [C_DIR] [varchar] (3) NULL ,
    [C_FLAG] [varchar] (1) NULL ,
    [D_LOG] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL ,
    [F_CQMS] [varchar] (1) NULL ,
    [staarchive] [varchar] (1) NULL
)
GO


CREATE TABLE [mil].[unit_auth] (
    [n_user] [varchar] (8) NOT NULL ,
    [unit] [varchar] (1) NOT NULL ,
    [plt] [varchar] (1) NOT NULL
) ON [PRIMARY]
GO


CREATE TABLE [mil].[ARMYCOMP] (
    [COMP] [varchar] (3) NOT NULL ,
    [XX] [varchar] (2) NULL ,
    [E_COMP] [varchar] (45) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL ,
    [lineid] [numeric](18, 0) NULL
)
GO


CREATE TABLE [mil].[ARMYUNIT] (
    [UNIT] [varchar] (1) NOT NULL ,
    [XX] [varchar] (2) NULL ,
    [N_UNIT] [varchar] (30) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL ,
    [lineid] [numeric](18, 0) NULL
)
GO
```

```
CREATE TABLE [mil].[M_STAT] (
    [MAR_STAT] [varchar] (1) NOT NULL ,
    [XX] [varchar] (2) NULL ,
    [MS_DESC] [varchar] (22) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL
)
GO


CREATE TABLE [mil].[QUOTA] (
    [QS1] [varchar] (2) NOT NULL ,
    [QS2] [varchar] (2) NULL ,
    [XX] [varchar] (2) NULL ,
    [QUOTA] [varchar] (45) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL
)
GO


CREATE TABLE [mil].[RACE] (
    [RACE_CODE] [varchar] (1) NOT NULL ,
    [XX] [varchar] (2) NULL ,
    [RACE] [varchar] (10) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL
)
GO


CREATE TABLE [mil].[SERVICE] (
    [SERVICE] [varchar] (1) NOT NULL ,
    [N_SERV] [varchar] (20) NULL ,
    [SERV_ABBR] [varchar] (4) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL
)
GO


CREATE TABLE [mil].[SM_STAT] (
    [SM_STATUS] [varchar] (1) NOT NULL ,
    [XX] [varchar] (2) NULL ,
    [SM_STAT_DE] [varchar] (21) NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL
)
GO


CREATE TABLE [mil].[ADMINBASE] (
    [BASD] [datetime] NULL ,
```

```
    [BPED] [datetime] NULL ,
    [BRANCH] [varchar] (2) NULL ,
    [CLASS] [varchar] (12) NULL ,
    [COMP] [varchar] (3) NULL ,
    [DOB] [datetime] NULL ,
    [D_ARRIVAL] [datetime] NULL ,
    [D_DEPART] [datetime] NULL ,
    [D_DLAB] [datetime] NULL ,
    [D_RANK] [datetime] NULL ,
    [D_UNIT] [datetime] NULL ,
    [ETS] [datetime] NULL ,
    [LIC_1] [varchar] (2) NULL ,
    [LIC_2] [varchar] (2) NULL ,
    [MAR_STAT] [varchar] (1) NULL ,
    [MEALCARD] [varchar] (9) NULL ,
    [N_STUDENT] [varchar] (27) NULL ,
    [PAY_CODE] [varchar] (2) NULL ,
    [PAY_GRD] [varchar] (2) NULL ,
    [PLT] [varchar] (2) NOT NULL ,
    [PMOS] [varchar] (9) NULL ,
    [Q_DLAB] [smallint] NULL ,
    [QS1] [varchar] (2) NULL ,
    [RACE] [varchar] (1) NULL ,
    [RANK] [varchar] (6) NULL ,
    [RMKS1] [varchar] (20) NULL ,
    [SERVICE] [varchar] (1) NULL ,
    [SEX] [varchar] (1) NULL ,
    [SM_STATUS] [varchar] (1) NULL ,
    [SQD] [varchar] (2) NULL ,
    [SSN] [varchar] (9) NOT NULL ,
    [ULT_MOS] [varchar] (9) NULL ,
    [UNIT] [varchar] (1) NULL ,
    [Q_AGE] [real] NULL ,
    [D_TRAN] [datetime] NULL ,
    [N_USER] [varchar] (8) NULL ,
    [PlaceOfBirth] [varchar] (30) NULL ,
    [d_ArrvOnPost] [datetime] NULL ,
    [t_ArrvOnPost] [datetime] NULL ,
    [d_EstArrv] [datetime] NULL ,
    [UnitDepart] [varchar] (40) NULL ,
    [InterimBillet] [varchar] (255) NULL ,
    [LangToStudy] [varchar] (40) NULL ,
    [Inprocessed] [varchar] (1) NULL ,
    [n_user_arrv] [varchar] (16) NULL ,
    [d_tran_arrv] [datetime] NULL ,
    [rec_stat] [varchar] (1) NULL
)
GO

ALTER TABLE [mil].[CLASS] WITH NOCHECK ADD
   CONSTRAINT [PK_CLASS] PRIMARY KEY  CLUSTERED
   (
      [C_CLASS]
```

```
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[ARMYCOMP] WITH NOCHECK ADD
    CONSTRAINT [PK_ARMYCOMP] PRIMARY KEY  CLUSTERED
    (
        [COMP]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[ARMYUNIT] WITH NOCHECK ADD
    CONSTRAINT [PK_ARMYUNIT] PRIMARY KEY  CLUSTERED
    (
        [UNIT]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[M_STAT] WITH NOCHECK ADD
    CONSTRAINT [PK_M_STAT] PRIMARY KEY  CLUSTERED
    (
        [MAR_STAT]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[QUOTA] WITH NOCHECK ADD
    CONSTRAINT [PK_QUOTA] PRIMARY KEY  CLUSTERED
    (
        [QS1]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[RACE] WITH NOCHECK ADD
    CONSTRAINT [PK_RACE] PRIMARY KEY  CLUSTERED
    (
        [RACE_CODE]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[SERVICE] WITH NOCHECK ADD
    CONSTRAINT [PK_SERVICE] PRIMARY KEY  CLUSTERED
    (
        [SERVICE]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[SM_STAT] WITH NOCHECK ADD
    CONSTRAINT [PK_SM_STAT] PRIMARY KEY  CLUSTERED
```

```
    (
        [SM_STATUS]
    )  ON [PRIMARY]
GO

ALTER TABLE [mil].[ADMINBASE] WITH NOCHECK ADD
    CONSTRAINT [PK_ADMINBASE] PRIMARY KEY  CLUSTERED
    (
        [SSN]
    )  ON [PRIMARY]
GO


ALTER TABLE [mil].[ADMINBASE] WITH NOCHECK ADD
    CONSTRAINT [DF_ADMINBASE_PLT] DEFAULT (' ') FOR [PLT],
    CONSTRAINT [DF_ADMINBASE_D_TRAN_1__14] DEFAULT (getdate()) FOR
[D_TRAN],
    CONSTRAINT [DF_ADMINBASE_N_USER_2__14] DEFAULT (user_name(null)) FOR
[N_USER],
    CONSTRAINT [DF_ADMINBASE_rec_stat_3__14] DEFAULT ('A') FOR
[rec_stat]
GO


 CREATE  INDEX [i_unitplt] ON [mil].[unit_auth]([unit], [plt]) ON
[PRIMARY]
GO


 CREATE  INDEX [i_unitplt] ON [mil].[ADMINBASE]([UNIT], [PLT]) ON
[PRIMARY]
GO


ALTER TABLE [mil].[ADMINBASE] ADD
    CONSTRAINT [FK_ADMINBAS_REF_1672_CLASS] FOREIGN KEY
    (
        [CLASS]
    ) REFERENCES [mil].[CLASS] (
        [C_CLASS]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_1675_ARMYUNIT] FOREIGN KEY
    (
        [UNIT]
    ) REFERENCES [mil].[ARMYUNIT] (
        [UNIT]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_1678_ARMYCOMP] FOREIGN KEY
    (
        [COMP]
    ) REFERENCES [mil].[ARMYCOMP] (
        [COMP]
    ),
```

```
    CONSTRAINT [FK_ADMINBAS_REF_1691_QUOTA] FOREIGN KEY
    (
       [QS1]
    ) REFERENCES [mil].[QUOTA] (
       [QS1]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_1694_SERVICE] FOREIGN KEY
    (
       [SERVICE]
    ) REFERENCES [mil].[SERVICE] (
       [SERVICE]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_1697_RACE] FOREIGN KEY
    (
       [RACE]
    ) REFERENCES [mil].[RACE] (
       [RACE_CODE]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_1700_M_STAT] FOREIGN KEY
    (
       [MAR_STAT]
    ) REFERENCES [mil].[M_STAT] (
       [MAR_STAT]
    ),
    CONSTRAINT [FK_ADMINBAS_REF_5470_SM_STAT] FOREIGN KEY
    (
       [SM_STATUS]
    ) REFERENCES [mil].[SM_STAT] (
       [SM_STATUS]
    )
GO


CREATE VIEW mil.ADMIN AS

SELECT BASD, BPED, BRANCH, CLASS, COMP, DOB, D_ARRIVAL,
    D_DEPART, D_DLAB, D_RANK, D_UNIT, ETS, LIC_1, LIC_2,
    MAR_STAT, MEALCARD, N_STUDENT, PAY_CODE, PAY_GRD,
    PLT, PMOS, Q_DLAB, QS1, RACE, RANK, RMKS1, SERVICE,
    SEX, SM_STATUS, SQD, SSN, ULT_MOS, UNIT, Q_AGE,
    D_TRAN, N_USER, rec_stat, PlaceOfBirth
   FROM mil.ADMINBASE
      WHERE EXISTS
         (SELECT *
            FROM mil.unit_auth u
            WHERE mil.ADMINBASE.unit + mil.ADMINBASE.plt LIKE u.unit
                  + u.plt AND u.n_USER = USER)


CREATE VIEW mil.admin_milx AS
 SELECT mil.ADMINBASE.basd,
```

```
        mil.ADMINBASE.bped,
        mil.ADMINBASE.branch,
        mil.ADMINBASE.class,
        mil.ADMINBASE.comp,
        mil.ADMINBASE.dob,
        mil.ADMINBASE.d_arrival,
        mil.ADMINBASE.d_depart,
        mil.ADMINBASE.d_dlab,
        mil.ADMINBASE.d_rank,
        mil.ADMINBASE.d_unit,
        mil.ADMINBASE.ets,
        mil.ADMINBASE.lic_1,
        mil.ADMINBASE.lic_2,
        mil.ADMINBASE.mar_stat,
        mil.ADMINBASE.mealcard,
        mil.ADMINBASE.n_student,
        mil.ADMINBASE.pay_code,
        mil.ADMINBASE.pay_grd,
        mil.ADMINBASE.plt,
        mil.ADMINBASE.pmos,
        mil.ADMINBASE.q_dlab,
        mil.ADMINBASE.qs1,
        mil.ADMINBASE.race,
        mil.ADMINBASE.rank,
        mil.ADMINBASE.rmks1,
        mil.ADMINBASE.service,
        mil.ADMINBASE.sex,
        mil.ADMINBASE.sm_status,
        mil.ADMINBASE.sqd,
        mil.ADMINBASE.ssn,
        mil.ADMINBASE.ult_mos,
        mil.ADMINBASE.unit,
        mil.ADMINBASE.q_age,
        mil.ADMINBASE.d_tran,
        mil.ADMINBASE.n_USER,
        mil.adminbase.placeofbirth,
        mil.ADMINBASE.rec_stat
   FROM mil.ADMINBASE
   WHERE EXISTS (SELECT * FROM mil.unit_auth u
                 WHERE mil.ADMINBASE.unit+mil.ADMINBASE.plt
                 LIKE u.unit+u.plt AND u.n_USER = USER)


CREATE VIEW  mil.apft AS
SELECT *
   FROM MIL.apftbase
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE MIL.APFTBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.ART15 AS
```

```
SELECT *
   FROM MIL.ART15BASE ART15BASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE ART15BASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.AWOL AS
SELECT *
   FROM MIL.AWOLBASE AWOLBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN W
                 HERE AWOLBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.BAR AS
SELECT *
   FROM MIL.BARBASE BARBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE BARBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.BRKS_ACT AS
SELECT *
   FROM mil.BRKS_ACTBASE
   WHERE EXISTS (SELECT *
                 FROM mil.unit_auth u
                 WHERE mil.BRKS_ACTBASE.company = u.unit AND u.n_USER
= USER)


CREATE VIEW  mil.CC_TRNG AS
SELECT *
   FROM MIL.CC_TRNGBASE CC_TRNGBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE CC_TRNGBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.CHAP AS
SELECT *
FROM MIL.CHAPBASE CHAPBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE CHAPBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.DEPN AS
```

```
SELECT *
   FROM MIL.DEPNBASE DEPNBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE DEPNBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.FCARE AS
SELECT *
   FROM MIL.FCAREBASE FCAREBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE FCAREBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.FLAG AS
SELECT *
   FROM MIL.FLAGBASE FLAGBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE FLAGBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.FORM90 AS
SELECT *
   FROM MIL.FORM90BASE FORM90BASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE FORM90BASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.ISSR AS
SELECT *
   FROM MIL.ISSRBASE ISSRBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE ISSRBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.LOC AS
SELECT *
   FROM MIL.LOCBASE LOCBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE LOCBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.LPTH AS
```

```
SELECT *
   FROM MIL.LPTHBASE LPTHBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE LPTHBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.lrt_plan AS
SELECT *
   FROM MIL.lrt_planBASE lrt_planBASE
   WHERE lrt_planBASE.C_UNIT IN (SELECT MIL.UNIT_AUTH.UNIT
                                 FROM  MIL.UNIT_AUTH
                                 WHERE MIL.UNIT_AUTH.N_USER=USER)


CREATE VIEW  mil.M_TRNG AS
SELECT *
   FROM MIL.M_TRNGBASE M_TRNGBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE M_TRNGBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.OFFICE AS
SELECT *
   FROM MIL.OFFICEBASE OFFICEBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE OFFICEBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.PREG AS
SELECT *
   FROM MIL.PREGBASE PREGBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE PREGBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.PRM_PRTY AS
SELECT *
   FROM MIL.PRM_PRTYBASE PRM_PRTYBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE PRM_PRTYBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.PROF AS
```

```
SELECT *
   FROM MIL.PROFBASE PROFBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE PROFBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.TRNG AS
SELECT *
   FROM MIL.TRNGBASE TRNGBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE TRNGBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.URINE AS
SELECT *
   FROM MIL.URINEBASE URINEBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE URINEBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW mil.v_cansee AS
SELECT * FROM mil.unit_auth
   WHERE n_USER=USER


CREATE VIEW  mil.WC AS
SELECT *
   FROM MIL.WCBASE WCBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE WCBASE.SSN=MIL.ADMIN.SSN)


CREATE VIEW  mil.wpn AS
SELECT *
   FROM MIL.wpnBASE wpnBASE
   WHERE EXISTS( SELECT *
                 FROM MIL.ADMIN
                 WHERE wpnBASE.SSN=MIL.ADMIN.SSN)
```

```
CREATE PROCEDURE mil.changessn ( @oldssn varchar(9), @newssn
varchar(9))AS

SELECT * INTO #tempbaseadmin
      FROM mil.adminbase
      WHERE mil.adminbase.ssn = @oldssn

UPDATE #tempbaseadmin
      SET ssn = @newssn, n_USER = USER, d_tran = GETDATE()
      WHERE ssn = @oldssn

INSERT INTO mil.adminbase
      SELECT * FROM #tempbaseadmin

UPDATE mil.GRAM                                SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.HHQ                                 SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CC_TRNGBASE                         SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.APFTBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ISSRMASTBASE                        SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LOI                                 SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ART15BASE                           SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.AWOLBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BARBASE                             SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BRKS_ACT                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CHAPBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.OFFICEBASE                          SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.DEPNBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PROFILES                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FCAREBASE                           SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FLAGBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FORM90BASE                          SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ISSRBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LOCBASE                             SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LPTHBASE                            SET ssn = @newssn WHERE ssn
= @oldssn
```

```
UPDATE mil.M_TRNGBASE                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PREGBASE                      SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.SEC                           SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PRM_PRTYBASE                  SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PROFBASE                      SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BAR1                          SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.REPRBASE                      SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.RESRV                         SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TRNG                          SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CO_FEED                       SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.wpnBASE                       SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TARGET                        SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CST                           SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TRNGMSTR                      SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.URINEBASE                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.VEH                           SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.WCBASE                        SET ssn = @newssn WHERE ssn
= @oldssn

DELETE FROM mil.adminbase
      WHERE mil.adminbase.ssn = @oldssn

GO


CREATE PROCEDURE mil.changeviewssn ( @oldssn varchar(9), @newssn
varchar(9))AS

SELECT * INTO #tempviewadmin
      FROM mil.admin
      WHERE mil.admin.ssn = @oldssn

UPDATE #tempviewadmin
      SET ssn = @newssn, n_USER = USER, d_tran = GETDATE()
      WHERE ssn = @oldssn

INSERT INTO mil.adminbase
      SELECT * FROM #tempviewadmin
```

```
UPDATE mil.GRAM                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.HHQ                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CC_TRNG                 SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.APFT                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ISSR                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LOI                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ART15                   SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.AWOL                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BAR                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BRKS_ACT                SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CHAP                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.OFFICE                  SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.DEPN                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PROFILES                SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FCARE                   SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FLAG                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.FORM90                  SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.ISSR                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LOC                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.LPTH                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.M_TRNG                  SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PREG                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.SEC                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PRM_PRTY                SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.PROF                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.BAR1                    SET ssn = @newssn WHERE ssn
= @oldssn
```

```
UPDATE mil.REPR                                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.RESRV                                   SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TRNG                                    SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CO_FEED                                 SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.wpn                                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TARGET                                  SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.CST                                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.TRNGMSTR                                SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.URINE                                   SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.VEH                                     SET ssn = @newssn WHERE ssn
= @oldssn
UPDATE mil.WC                                      SET ssn = @newssn WHERE ssn
= @oldssn

DELETE FROM mil.admin
      WHERE mil.admin.ssn = @oldssn

GO


CREATE PROCEDURE mil.getRecordStatus ( @newssn varchar(9) )AS

declare @recstat varchar(1)

SELECT @recstat = rec_stat
      FROM mil.adminbase
      WHERE ssn = @newssn

return convert( int, @recstat)

GO


CREATE PROCEDURE mil.getSUID( @TABLE_NAME VARCHAR(384),
@TABLE_USER VARCHAR(384), @TABLE_PERMS VARCHAR(4)  OUTPUT)
AS

if ( @TABLE_NAME is null) OR ( @TABLE_USER is null)
     begin
     raiserror 20001 'Must provide table name AND USER ID.'
     return
end
```

```
DECLARE @sel char(1)
DECLARE @updt char(1)
DECLARE @insrt char(1)
DECLARE @dlt char(1)

SELECT @sel = '?'
SELECT @updt = '?'
SELECT @insrt = '?'
SELECT @dlt = '?'

SELECT @sel = 'S' FROM sysprotects p, sysobjects o, sysUSERs u,
sysmembers m
WHERE p.id = o.id
and o.type IN ('U','V','S') AND object_name(o.id) = @TABLE_NAME
and USER_name(u.uid) = @TABLE_USER
and (u.uid > 0 AND u.uid < 16384)
and ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
AND p.action = 193 /*SELECT*/

SELECT @updt = 'U' FROM sysprotects p, sysobjects o, sysUSERs u,
sysmembers m
WHERE p.id = o.id
and o.type IN ('U','V','S') AND object_name(o.id) = @TABLE_NAME
and USER_name(u.uid) = @TABLE_USER
and (u.uid > 0 AND u.uid < 16384)
and ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
AND p.action = 197 /*UPDATE*/

SELECT @insrt = 'I' FROM sysprotects p, sysobjects o, sysUSERs u,
sysmembers m
WHERE p.id = o.id
and o.type IN ('U','V','S') AND object_name(o.id) = @TABLE_NAME
and USER_name(u.uid) = @TABLE_USER
and (u.uid > 0 AND u.uid < 16384)
and ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
AND p.action = 195 /*insert*/

SELECT @dlt = 'D' FROM sysprotects p, sysobjects o, sysUSERs u,
sysmembers m
WHERE p.id = o.id
and o.type IN ('U','V','S') AND object_name(o.id) = @TABLE_NAME
and USER_name(u.uid) = @TABLE_USER
and (u.uid > 0 AND u.uid < 16384)
and ((p.uid = u.uid) OR (p.uid = m.groupuid AND u.uid = m.memberuid))
AND p.action = 196 /*delete*/

SELECT @TABLE_PERMS = @sel + @updt + @insrt + @dlt

GO
```

```
CREATE PROCEDURE mil.isInAdminbase( @newssn varchar(9), @recstat
varchar(1) OUTPUT) AS

SELECT @recstat = '?'

SELECT @recstat = rec_stat
      FROM mil.adminbase
      WHERE ssn = @newssn

GO
```

INTITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Commanders.....1
   DLIFLC & POM
   ATZP-GC
   (Attn: COL Dausen)
   Presidio of Monterey, CA 93944

4. Commanders.....1
   DLIFLC & POM
   ATZP-IM
   (Attn: Winnie Chambliss)
   Presidio of Monterey, CA 93944

5. Commanders.....1
   DLIFLC & POM
   ATZP-IM-AT
   (Attn: Kristina Brown)
   Presidio of Monterey, CA 93944

6. Commanders.....1
   DLIFLC & POM
   ATZP-IM-KS
   (Attn: Pat Golden)
   Presidio of Monterey, CA 93944

7. Chairman, Code CS.....1
   Naval Postgraduate School
   1 University Circle
   Monterey, CA 93943-5101

8. Dr. Thomas C. Wu, Code CS/Wu.....1
   Naval Postgraduate School
   1 University Circle
   Monterey, CA 93943-5101

9. LCDR Chris Eagle, Code CS/Ce.....1
Naval Postgraduate School
1 University Circle
Monterey, CA 93943-5101

10. Pavel Bielecki.....1
500 Glenwood Cr., STE 535
Monterey, CA 93940